MASTER EN SCIENCES INFORMATIQUES

La table des symboles du compilateur
Essai de synthèse des problèmes relatifs à sa construction et à sa consultation

Lesuisse, Roland

Award date:
1974

Awarding institution:
Universite de Namur

Editeur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR

Institut d'Informatique

Année académique 1973-1974

LA TABLE DES SYMBOLES DU COMPILATEUR

ESSAI DE SYNTHESE DES PROBLEMES RELATIFS A SA CONSTRUCTION ET A SA CONSULTATION.

Roland LESUISSE

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique.

A Jacqueline,

A Olivier et Sandrine.

Notre plus grand merci à Monsieur le Professeur LEROY qui a dirigé ce mémoire avec rigueur et bienveillance.

Merci à tous ceux qui, de mille manières différentes, ont encouragé et favorisé notre "reconversion".

TABLE DES MATIERES

Pages

Pages

Pages

—

1. INTRODUCTION

Puisque notre but avoué vise à traiter des problèmes relatifs à la Table des Symboles du compilateur, il nous semble souhaitable de définir d'abord la place du processus de compilation dans un processus informatique et de déterminer ensuite le rôle qu'y joue la Table des Symboles. Ce sera l'objet de cette introduction.

1.1. Machine et langage

1.1.1. Problème et automate

Pour qu'un problème puisse être mécaniquement traité, il importe que trois conditions, au moins, soient préalablement réunies :

- que le problème existe;
- qu'on découvre un algorithme conduisant à la solution, c'est-à-dire "un ensemble de règles qui disent, de moment à moment et avec précision, comment se comporter" (1).
  Naturellement ces règles sont exprimées dans un langage préalablement décrit;
- qu'il existe "un automate simple" - c'est-à-dire un système logique sans pouvoir d'innovation ni d'intelligence, capable d'interpréter les instructions écrites dans le langage et donc d'exécuter les pas de chaque processus spécifié" (1).

Sur une machine réelle, l'algorithme prendra alors le nom de programme; le langage dans lequel est décrit l'algorithme exécutable deviendra le langage-machine dont voici les principales caractéristiques :

1) l'alphabet sur lequel il est défini se réduit à l'ensemble des symboles de l'algèbre de Boole. Pour la facilité, nous le désignerons par $\{0,1\}$;

---

(1) MINSKY, Computation, p. 106.

2) sa syntaxe correspond au format des instructions-machines. Par exemple, une instruction-machine (chaîne de caractères binaires $\in$ langage-machine) aura une longueur de 32 positions binaires; les 8 premières positions à gauche représenteront le code opératoire, dont la valeur devra appartenir à l'ensemble des valeurs acceptables, tandis que les 24 dernières positions contiendront une référence à une adresse-mémoire;

3) sa sémantique décrit le comportement de la machine réelle.

Enfin, l'automate simple capable d'interpréter le langage-machine porte généralement le nom de Central Processor Unit (C.P.U.).

1.1.2. Inadéquation du langage machine et solutions (1)

Sur les anciennes machines, tout le codage était fait en langage-machine. Il en résultait que le programmeur devait porter essentiellement son attention sur l'écriture du programme et que, en conséquence, la logique du problème traité devait être pratiquement (2) simple.

Rien d'étonnant, dès lors, à ce que les constructeurs se soient, très tôt, efforcés de réduire cette barrière linguistique. La solution la plus naturelle consistait à écrire en langage-machine "un programme qui accepterait comme donnée une espèce de langage symbolique et le traduirait en langage-machine" (3).

Dans un premier temps, le programme traducteur fut appelé un "assembleur".

Le langage-donnée présentait deux caractéristiques importantes : il était symbolique et isomorphe au langage-machine.

---

(1) Pour ce passage, voir INGERMAN, Translator, pp. 3 - 5.

(2) Il est obvie, en effet, que la limite de la complexité n'est pas d'ordre théorique. Sans quoi, on aboutirait à la proposition aberrante : "la logique d'un programme écrit en FORTRAN peut être plus complexe que celle d'un programme écrit en langage-machine" ?

(3) INGERMAN, Translator, p. 2.

Symbolique, il rendait plus aisée la tâche du programmeur et autorisait, pratiquement, le traitement de problèmes à logique complexe (1); isomorphe au langage-machine, il restait, en fait, relatif à une machine.
Cette caractéristique allait, dans le développement des applications informatiques, s'avérer fort gênante.

Aussi, l'étape suivante vit-elle la naissance de langages libérés de cette contrainte d'isomorphisme; en outre, leur forme plus "naturelle" permit aux concepteurs et aux analystes d'écrire leurs programmes sans intermédiaire.

La machine s'était donc progressivement rapprochée de certaines exigences humaines. En contrepartie, la logique du programme-traducteur, devenu compilateur, s'accrut dans des proportions identiques.

1.2. Le compilateur et le processus de compilation

1.2.1. Définition d'un compilateur

Nous inspirant de Post, nous pourrions concevoir le compilateur comme "un ensemble de règles qui disent comment certaines chaînes de symboles peuvent être transformées en d'autres chaînes de symboles." (2). Mais en dernière analyse, pareille définition s'appliquerait, sans difficulté, à tout système mathématique ou logique, même le plus puissant.

Aussi dirons-nous plus précisément qu'un compilateur est un système logique qui accepte comme entrée un message écrit dans un langage (LANGAGE SOURCE) et produit, à la sortie, un message écrit dans un autre langage (LANGAGE CIBLE ou OBJET), sous

---

(1) On s'en convaincra, en se souvenant que certaines méthodes d'analyse, en informatique de gestion, s'appuient sur le langage symbolique d'assemblage.

(2) E.L. POST, Formal reductions of the general combinatorial decision problem, American Journal of Mathematics 65, 197 - 288 in MINSKY, Computation, p. 219.

la contrainte que les deux messages doivent avoir une seule et même signification" (1). Les messages d'entrée et de sortie auront, en général, une syntaxe et une sémantique différente. (2)

Essentiellement, le compilateur est donc un traducteur.

1.2.2. Processus de compilation (3)

De façon générale, un processus de compilation est divisé logiquement en deux phases, comme le montre notre schéma Fig. 1.2.2. :

1) Une phase de RECONNAISSANCE qui examine le langage Source et délivre, à la sortie, la structure de la phrase dans un langage intermédiaire (Polonais, postfixé, arbre ...)
   Pratiquement, cette procédure opère la conversion syntaxique entre phrases du langage Source et du langage Cible.

2) Une phase de COMMANDE (ou Génération) qui utilise cette structure de phrase et produit la forme traduite du texte initial.

---

(1) INGERMAN, Translator, p. 12.
Garvin soutenait, quoique avec moins de précision, une idée comparable. "Dans un processus de traduction, l'expression du contenu dans un langage est remplacée par l'expression d'un contenu équivalent dans un autre." GARVIN, Machine Translation, p. 29.

La notion d'invariance de signification apparaît encore dans ce texte de "Quoique une fonction qui transforme des phrases d'un langage source dans un langage objet soit un traducteur, un traducteur efficace doit conserver la signification des phrases traduites." MC KEEMAN, Compiler generator, p. 38.
La distinction opérée ici entre traducteur et traducteur efficace nous paraît factice sinon erronée : traduire n'implique-t-il pas NECESSAIREMENT le maintien de l'invariance de signification ?

(2) INGERMAN, Translator, p. 12.

(3) Notre propos consiste uniquement à décrire la procédure généralement utilisée pour traduire en langage-machine, les langages informatiques tels qu'ils existent actuellement.

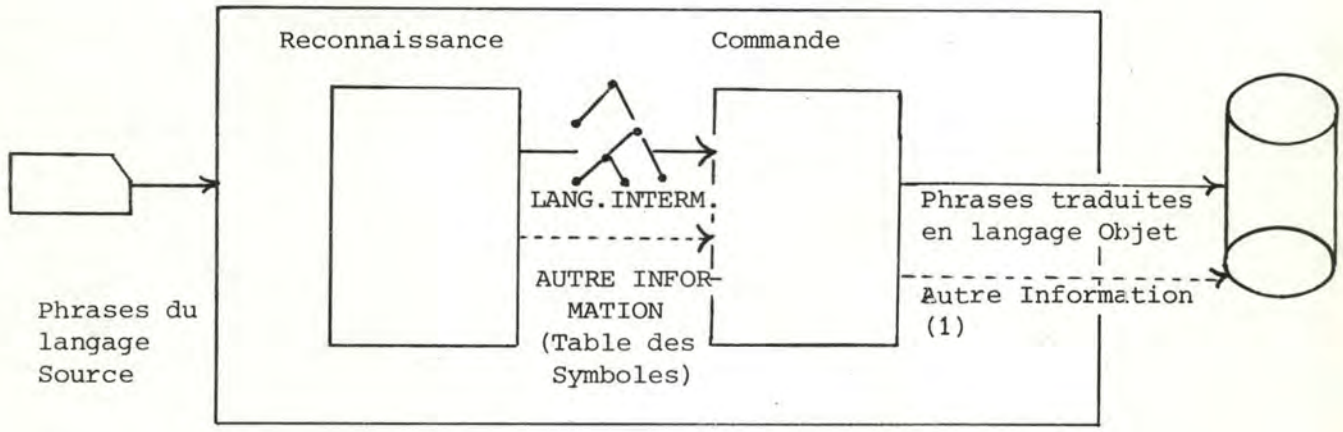


Fig. 1.2.2. Processus de compilation (2)

1.2.3. La nécessité de dictionnaires

Considérons, maintenant du point de vue d'un traducteur le processus décrit Fig. 1.2.2. Il apparaît clairement :

1) que l'exécution de la phase de reconnaissance réclame l'existence d'un dictionnaire qui associe à chaque morphème du Langage Source la classe grammaticale (3) à laquelle il appartient, ainsi qu'un algorithme de reconnaissance basé sur une grammaire du Langage Source;
2) que l'exécution de la phase de commande fait appel à un algorithme de "traduction" et à un dictionnaire bilingue qui à chaque morphème du Langage Source associe un sous-ensemble de morphèmes du Langage Objet.

Bien qu'à première vue, ils apparaissent fort distincts, les éléments de ces différents dictionnaires sont, en général, regroupés dans ce qu'il est convenu d'appeler la Table des Symboles du compilateur.

---

(1) Par exemple, l'information destinée au relieur.

(2) MC KEEMAN, Compiler Generator, p. 38.

(3) Dans un langage naturel, on distingue, par exemple, la classe des verbes, des noms, etc...

1.3. La Table des Symboles du Compilateur

1.3.1. Définition

Grossièrement parlant, la Table des Symboles se veut donc la représentation d'un dictionnaire. Cependant, si on se donne pour objectif d'en étudier forme et contenu, il importe, nous semble-t-il, d'appréhender plus finement la spécificité de cet objet informatique. A ce propos, nous éviterons un écueil désormais connu : une conception trop uniquement mathématique. Ainsi cette définition de Wegner :

> " Une Table de Symboles apparaît comme définissant une fonction (tabulaire) dont le domaine est l'ensemble S de tous les symboles d'entrée (arguments) et dont l'image est l'ensemble T de tous les attributs (valeurs) associés aux différents $s_i \in S$. Les Tables de Symboles conviennent (particulièrement) pour les spécifications de fonctions dont le domaine est fini, et les règles de correspondance entre arguments et valeurs sont souples." (1)

Cette définition, mathématiquement sans reproche, ne suffit point, cependant, à rendre compte d'un objet dont l'existence est, sans conteste, liée à un support d'information.

C'est pourquoi, dans le dessein de pallier la carence que nous venons de dénoncer, Hopgood écrit

> " qu'une Table est un dispositif de mémorisation pour des
> " items à chacun desquels est associé un nom unique ou
> " clé." (2)

---

(1) WEGNER, Programming Languages, pp. 112 et 120. Gries développe une idée analogue quand il distingue dans les entrées de sa Table la composante ARGUMENT et la composante VALEUR, GRIES, Compiler Construction, p. 213.

(2) HOPGOOD, Compiling Techniques, p. 16.

Cette approche est, sans nul doute, complète : les plans logique et physique y trouvent leur content. Nous lui préférons, pourtant, pour sa simplicité et sa netteté, la définition de Knuth : "Une table est un petit fichier." (1)

Montrons que cette façon de voir les choses prend bien en charge la structure logique et la structure physique de la Table des Symboles.

1.3.2. Justification de la définition de Knuth

En informatique, un fichier désigne un ensemble d'informations muni de certaines structures [...], les éléments structurants se rangeant en 3 catégories :

a) des relations de signification reliant entre elles les informations jugées indispensables à la résolution du problème posé. Lorsque nous les aurons établies, nous disposerons donc des données.

b) des relations permettant l'accès à une information du fichier. En effet, les données, une fois rangées dans quelque dispositif de mémorisation que ce soit, il importe que nous puissions univoquement les retrouver.

c) des relations d'implémentation physique. Il convient, en effet, que l'information, conceptuellement rassemblée par les relations sémantiques et à laquelle on peut désormais conceptuellement accéder grâce aux relations d'accès, soit rangée sur un support. Ce qui postule que nous choisissions une unité de mémorisation (Mémoire Centrale, Disque Magnétique ...), et la manière selon laquelle l'information y sera rangée (Blocage, etc.). (2)

Par surcroît, cette démarche dégage, quasi naturellement, une séquence rigoureuse d'étapes à suivre lors de la création et de

---

(1) KNUTH, Art of Programming, III, p. 389.

(2) CHERTON, Cours, pp. I, 1 - I,2, passim.

la consultation d'une Table de Symboles et, en substance, nous nous y conformerons strictement.

Cependant, à l'expérience, nous nous sommes aperçu que le fichier que nous créions avait une nature particulière : son taux de consultation très élevé et sa taille relativement modeste faisaient que la Mémoire Centrale se révélait être son support adéquat. En outre, aucun facteur de blocage n'intervenait.

Ainsi, les relations d'implémentation devenaient si étroitement liées aux relations d'accès qu'il nous aurait paru factice d'en distinguer la présentation dans des sections différentes de notre travail.

C'est pourquoi, nous allons successivement envisager - comme il est de tradition en la matière (1) - la description des données de la Table des Symboles du Compilateur - ce sera l'objet du deuxième chapitre - et la recherche des informations dans la Table des Symboles, que nous traiterons dans le dernier chapitre.

---

(1) Cfr. GRIES, Compiler Construction, ch. 9 et 10; HOPGOOD, Compiling Techniques, pp. 16 - 29, etc...

2. DESCRIPTION DES DONNEES

Dans ce chapitre, nous nous proposons d'abord de déterminer, indépendamment de tout langage et de toute méthode particulière de compilation, le type général d'informations requis dans une Table de Symboles; ensuite, nous appliquerons les résultats obtenus au langage ALGOL 60; enfin, nous aborderons le problème particulier des variables structurées (le seul problème de quelque importance laissé dans l'ombre par ALGOL 60.)

2.1. Type général d'informations dans une Table de Symboles

Reprenons, pour le préciser, le processus de compilation, tel que nous l'avons décrit (1) et essayons d'en déduire les éléments obligés de toute Table de Symboles.

2.1.1. Phase de reconnaissance

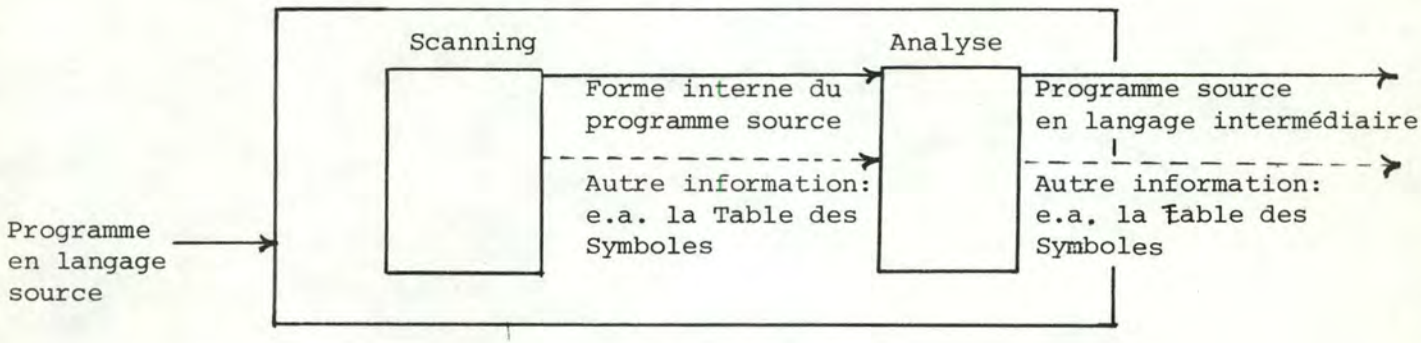


Fig. 2.1.1. (2)

. Le SCANNER a pour tâche minimale de lire les caractères du programme écrit en langage source et, tirant profit des signes spéciaux (blancs, signes de ponctuation), de reconstituer morphèmes et phrases qu'il propose ensuite à l'analyseur.

---

(1) Cfr. pp.4-5

(2) Mc KEEMAN, Compiler Generator, p. 39

. L'ANALYSEUR se voit, en général, assigner deux missions :

1) Vérifier la correction syntaxique du Programme Source
A cet effet, il se doit de connaître, pour chacun des morphèmes utilisés, la classe grammaticale à laquelle ce morphème appartient. Or, a priori, seuls les morphèmes réservés du langage source (par exemple en ALGOL 60, begin, for ...) sont, du point de vue de l'analyseur, complètement définis. C'est pourquoi, avant tout traitement, il est nécessaire que chaque morphème inconnu déclare la classe grammaticale à laquelle il appartient ainsi que ses caractéristiques propres (1). Ces renseignements seront rangés dans la Table des Symboles ainsi construite. Les informations à présent rangées dans la Table correspondent aux informations du premier dictionnaire (cfr. p.5)
En pratique, fréquemment, le rôle du SCANNER consiste aussi à remplacer les morphèmes qu'il a reconnus par des références à une table des symboles. Dans pareille optique, il est obvie que la construction de cette Table incombe au SCANNER.

2) Déclencher les procédures de réarrangement syntaxique (ce que Gries appelle les routines sémantiques). Grossièrement décrit, ce mécanisme a pour objet de réarranger les symboles du Programme-Source dans un ordre proche de celui où ils apparaîtront dans le Programme-Objet. Le résultat de ces routines sera, par exemple, la mise sous forme polonaise postfixée du Programm- Source.

2.1.2. Phase de Commande

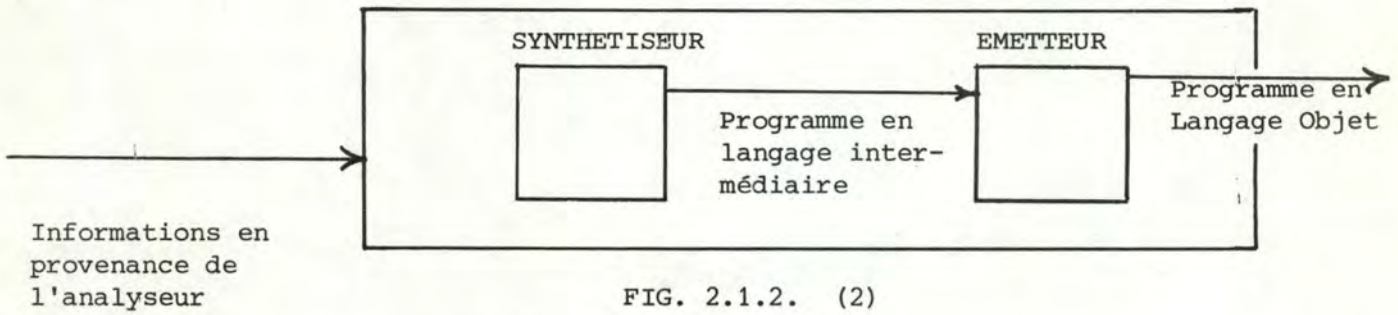


FIG. 2.1.2. (2)

(1) "La déclaration d'un identificateur sert à définir certaines propriétés des quantités utilisées dans le Programme et à les associer à des identificateurs". Rapport ALGOL 60, p. 3.

(2) Mc KEEMAN, Compiler Generator, p. 39

. Le SYNTHETISEUR a pour rôle d'opérer une traduction logique, entendons par là qu'elle ne tient pas compte, par exemple, des limites des registres dans le transfert des Zones - Mémoire.
Ce processus doit donc, entre autres, établir à tout moment t de l'exécution d'un programme, une bijection entre le sous-espace des noms actifs à ce moment-là (les identificateurs accessibles) et un sous-espace de l'espace des adresses/mémoire. Aussi liera-t-il tout identificateur à une adresse dont la signification différera selon les classes grammaticales. En fait, le synthétiseur complète le dictionnaire bilingue dont nous avons parlé (1).

. L'EMETTEUR adaptera le texte produit par le SYNTHETISEUR aux exigences de la machine.

En résumé, et de manière générale, le SCANNER construit la Table des Symboles;

l'ANALYSEUR y consigne les déclarations ;

le SYNTHETISEUR y mémorise une adresse/mémoire.

A cette étape logique, la Table des Symboles du compilateur n'a plus de raison d'exister.

2.2. Application au langage ALGOL 60

2.2.1. Pourquoi choisir ALGOL 60 ?

Pour deux raisons fort simples :

1) le problème de la compilation s'y pose dans les termes les plus généraux (structure de blocs, gestion dynamique de la mémoire qui en découle), ce à quoi ne peuvent naturellement prétendre ni FORTRAN, ni COBOL.
Seule fait défaut la déclaration des variables structurées que nous examinerons dans un paragraphe suivant.

---

(1) Cfr. p. 5

2) Le processus de compilation de ce langage est bien connu et largement publié (1) à la différence d'un langage plus puissant tel PL/1.

2.2.2. La Collecte de l'Information

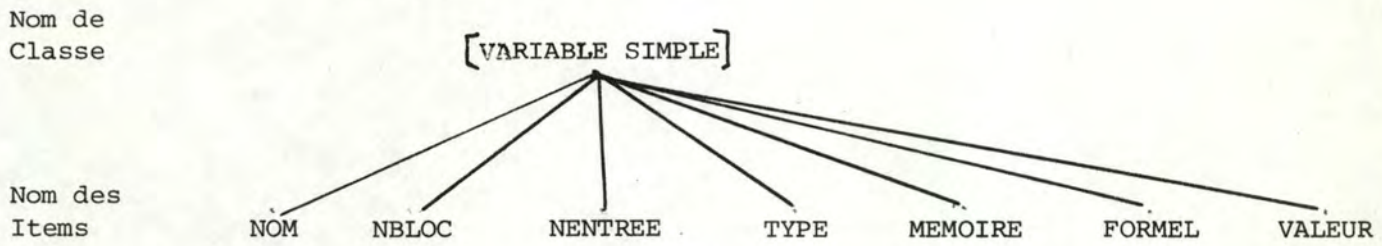
Dorénavant, nous ne nous intéresserons plus aux mots réservés du langage, ni aux constantes, pour reporter toute notre attention sur les identificateurs d'un programme ALGOL 60, c'est-à-dire sur "des objets" qui n'ont aucune signification inhérente mais servent à l'identification de tableaux, étiquettes, switches et procédures." (2).
De là apparaît notre notion de classe grammaticale (Quantities dans le Rapport ALGOL 60), essentielle du point de vue de l'analyseur syntaxique.
Nous obtiendrons donc une première partie de l'information qui nous est nécessaire, si nous parvenons à associer chaque classe grammaticale à l'ensemble des propriétés possibles d'un de ses éléments, autrement dit, si nous réussissons à décrire l'ensemble des déclarations syntaxiquement correctes dans un programme écrit en ALGOL 60.
Nous disposerons de toute l'information minimale (3) requise, si nous y ajoutons les différents renseignements apportés par les modules logiques du programme compilateur (scanner, analyseur, synthétiseur), selon le schéma général décrit Fig. 2.1.2.
Dès lors, notre manière de travailler semble toute tracée: nous allons pour chacune des quatre grandes classes grammaticales (4) : variable simple , tableau, procédure et étiquette, établir sous forme de graphe élémentaire les différentes propriétés possibles d'un de leurs éléments; ensuite,pour chaque item retenu, nous donnerons, dans un tableau explicatif, les valeurs qu'il peut prendre et - à titre d'exemple (5) - le rôle qu'il peut jouer dans le processus de compilation.

---

(1) Cfr. e.a. RANDEL, Implementation.- GRAU, Translation.
(2) Rapport ALGOL 60, p. 3.
(3) Toute autre information est fonction de la méthode de compilation et ne sera donc pas prise en compte dans le cadre de notre travail.
(4) Nous considérons qu'un switch se ramène à une procédure. Communication personnelle de J.P. Cardinael.
(5) Il est évident qu'il ne s'agit aucunement de justifier pourquoi nous avons retenu cet item : cela équivaudrait à justifier le langage lui-même.

2.2.2.1. Classe des variables simples

A. Graphe



B. Tableau explicatif

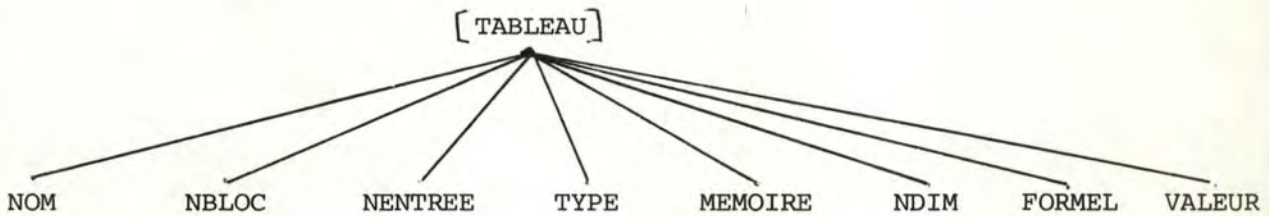
ITEMS	PLAGE DE VALEURS	EXPLICATION
NOM	Toute valeur qui obéit à la règle : ⟨identifier⟩::= ⟨letter⟩ ⟨identifier⟩⟨letter⟩ ⟨identifier⟩⟨digit⟩ Rapport ALGOL 60, p. 3	Cette notion, malgré ses apparences, est surtout liée à la structure d'accès à la Table. Il suffit, pour s'en convaincre, d'imaginer (cas idéal!) une Table de 2^k entrées et un ensemble S de 2^k identificateurs différents, chacun d'une longueur de K bits. Evidemment, il n'y aurait nul intérêt à ranger le nom de l'identificateur dans la Table, puisque chacun d'entre eux référerait à 1 et 1 seule entrée de la Table. En termes de fichier, nous parlerions alors d'ITEM VIRTUEL.
NBLOC	Numéro de niveau du bloc de la procédure . Les valeurs prises ε à l'{entier de 1 à k} si k désigne, pour un programme donné, le plus haut niveau d'imbrication atteint.	La nécessité de cet item apparaît si on se rappelle qu'en ALGOL 60, un identificateur est univoquement désigné par le couple (NOM, NBLOC).

ITEMS	PLAGE DES VALEURS	EXPLICATION
N ENTREE	Un entier $\in$ {des entiers de 1 à n} si n désigne la longueur logique de la Table [Nombre d'entrées]	C'est la façon dont l'identificateur est connu dans le Programme, dès qu'il a été pris en charge par le SCANNER.
TYPE	{Entier, réel, booléen}. Rien n'empêche de choisir une convention selon laquelle, par exemple, ENTIER = 1 REEL = 2 BOOLEEN= 3	Cet item permet, entre autres, 1) à l'analyseur de vérifier la correction syntaxique d'une phrase. Par ex. If ⟨expr. bool⟩ then ... 2) au synthétiseur de discriminer les calculs en virgule flottante des calculs en entiers ... et de générer les instructions-machine adéquates.
MEMOIRE	Supposons qu'à chaque bloc du programme soit affectée une zone de données. Alors, généralement, cette adresse sera fournie sous la forme (n,p) où n désigne le numéro de niveau du bloc [NBLOC] et p le déplacement relatif au début de la zone de données qui lui correspond.	cfr. p. 11 Rôle du Synthétiseur.
FORMEL	{Normal, Formel}, ou conventionnellement : NORMAL = 0 FORMEL = 1	Au cas où il s'agit d'un paramètre formel, l'analyseur syntaxique devra vérifier, lors d'un appel de procédure, la concordance entre la déclaration du paramètre actuel et la spécification du paramètre formel.

ITEMS	PLAGE DE VALEURS	EXPLICATION
VALEUR	Dans le cas d'un paramètre formel [Item précédent = 1], il faut distinguer le type de l'appel : NOM [= 0], VALEUR [= 1].	Les actions à générer dans le Programme-Objet diffèrent radicalement selon que le paramètre est appelé par nom ou par valeur. Cfr. GRIES, Compiler Construction, p.

2.2.2.2. Classe des Tableaux

A. Graphe

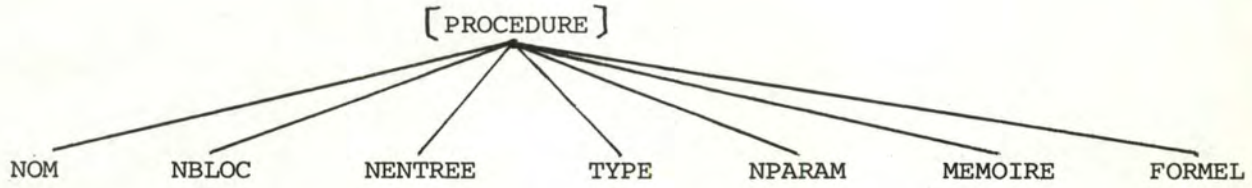


B. Tableau Récapitulatif

ITEMS	PLAGE DE VALEURS	EXPLICATION
NOM NBLOC NENTREE TYPE	Cfr. Classe des Variables Simples.	Cfr. Classe des Variables Simples.
MEMOIRE	Une référence à une entrée de la Table des Symboles.	L'implantation des tableaux en ALGOL 60 est essentiellement dynamique. Cette entrée référera donc, en toute généralité, aux paramètres d'adressage des variables indicées correspondantes.
NDIM	Une entrée sans limitation théorique.	Comment, sans ce renseignement, vérifier la correction de l'usage d'une variable indicée ?
FORMEL VALEUR	Cfr. Classe des Variables Simples.	Cfr. Classe des Variables Simples.

2.2.2.3. Classe des Procédures

A. Graphe

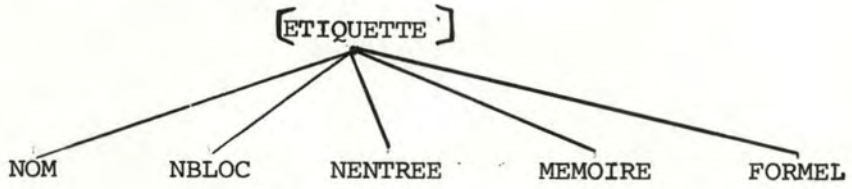


B. Tableau Explicatif

ITEMS	PLAGE DES VALEURS	EXPLICATIONS
NOM NBLOC NENTREE	Cfr. Classe des Variables	Cfr. Classe des Variables
TYPE	En fait, identique à 'variable', sauf qu'il faut y ajouter la valeur 'vide' (lorsqu'il ne s'agit pas d'une fonction).	Cfr. Classe des Variables
NPARAM	1 entier sans limitation théorique.	Un appel de procédure a-t-il le nombre de paramètres requis ?
MEMOIRE	1 adresse $\in$ {à l'espace des adresses}.	Il est essentiel, au moment d'un appel de procédure, de connaître à quel endroit du Programme-Objet l'exécution doit se poursuivre.
FORMEL	Cfr. Classe des Variables	Cfr. Classe des Variables

2.2.2.4. Classe des Etiquettes

A. Graphe



B. Tableau Explicatif

ITEMS	PLAGE DE VALEURS	EXPLICATION
NOM NBLOC NENTREE	Cfr. Classe des Variables	Cfr. Classe des Variables
MEMOIRE	1 adresse à l'espace des adresses.	Il faut savoir, lors d'une instruction de branchement, à quel endroit du Programme-Objet se brancher.
FORMEL	Cfr classe des variables	Cfr classe des variables

2.2.3. Dessin de l'Article Logique (1)

Il faut, à présent, intégrer dans un fichier les informations que nous avons, dans le cadre de notre problème, jugées nécessaires. Or, par essence, un fichier est destiné à être manipulé(lu ou écrit). Dès lors, ces informations doivent être rangées sur un support de telle façon qu'on puisse :

1) déterminer, avec précision, un ensemble d'informations qui forment un tout logique. Pareil ensemble s'appellera article logique, et nous nommerons enregistrement logique une occurrence d'un article logique.

---

(1) Logique car 1) aux informations que nous allons prendre en charge peuvent s'ajouter d'autres renseignements liés à la structure d'accès (par exemple, des pointeurs, des indications de longueur de zones...);
2) nous ne tenons pas compte actuellement des critères proprement physiques (longueur des items, rangement des enregistrements sur le support retenu ...).

Par exemple, {NOM, NENTREE, NBLOC, CLASSE GRAMMATICALE, MEMOIRE}$_{FORMEL}$ décrit l'article logique du fichier Table des Symboles correspondant à un identificateur d'étiquette (cfr p.17), tandis que {1, 17, 3, label, 78} décrirait l'enregistrement logique correspondant à l'identificateur 1;

2) localiser chaque atome d'un enregistrement logique, si du moins nous entendons par atome la valeur prise par un item de l'article logique. Ainsi, dans l'exemple précédent, il importe que nous puissions distinguer 'label' des atomes qui l'entourent (3 et 78). Cette opération est, en l'occurrence, réalisée grâce au séparateur ',';
3) reconnaître la nature de chaque atome, c'est-à-dire l'item dont il est une occurrence. Par exemple, il nous faut pouvoir associer à 17 l'item NENTREE.

En résumé, ces contraintes stipulent simplement que :

1) il faut adopter un ordre de rangement des items dans un article;
2) il faut se donner un moyen de déterminer, pour chaque enregistrement, la configuration d'article qui lui correspond si cette configuration n'est pas uniforme.

Ce problème de rangement de l'information accepte, dans le cas qui nous occupe, deux solutions que nous allons succinctement passer en revue.

2.2.3.1. Article dont le nombre et la nature des atomes est variable

A première vue, cette stratégie convient fort bien au problème posé et les différentes configurations de l'article de la Table des Symboles découlent, quasi naturellement, de la manière dont nous avons recueilli l'information.
Les voici, à titre d'exemple :

a) Configuration d'Article correspondant à un identificateur de VARIABLE SIMPLE

ITEMS	NOM	NBLOC	NENTREE	TYPE	MEMOIRE	FORMEL	VALEUR	CLASSE GRAM-MATICALE

b) Configuration d'Article correspondant à un identificateur de TABLEAU

ITEMS	NOM	NBLOC	NENTREE	TYPE	MEMOIRE	FORMEL	VALEUR	NDIM	CLASSE GRAM-MATICALE

c) Configuration d'Article correspondant à un identificateur de PROCEDURE

ITEMS	NOM	NBLOC	NENTREE	TYPE	MEMOIRE	NPARAM	CLASSE GRAM-MATICALE	FORMEL

d) Configuration d'Article correspondant à un identificateur d'ETIQUETTE

ITEMS	NOM	NBLOC	NENTREE	MEMOIRE	CLASSE GRAM-MATICALE	FORMEL

Avantages de cette technique : La place occupée sur le support, quel qu'il soit, est minimale. Mais cet avantage tend à s'amenuiser si :

1) les articles relativement les plus longs (par exemple, l'article correspondant à un identificateur de TABLEAU) sont, en valeur absolue, courts (par exemple, dans la Table des Symboles que nous construirions en Mémoire Centrale, l'article le plus long aurait une longueur de l'ordre de 2 mots/mémoire);

2) l'écart-type de la distribution des longueurs est petit, ce qui est bien le cas en l'occurrence, puisque seul l'article correspondant à un identificateur d'ETIQUETTE est sensiblement plus court que les autres.

Inconvénients de cette technique : La logique d'accès se révèle de beaucoup plus complexe.

En effet, lors de chaque accès, il convient de détecter, par un test de l'item CLASSE GRAMMATICALE, la configuration de l'enregistrement accédé.

C'est pourquoi, très généralement, la seconde solution, que nous allons décrire, a été préférée.

2.2.3.2. Article dont la nombre et la nature des 'ITEMS' est invariant.

Par sa logique, le problème ne se pose pas en ces termes-là, puisque, par exemple, à deux classes grammaticales différentes peut correspondre un nombre d'items différent. Il faut donc tâcher de s'y ramener, car l'avantage escompté, c'est-à-dire la standardisation de l'accès, apparaît considérable (1) en regard de l'inconvénient que représente la perte de place encourue, d'ailleurs minime du fait que les articles sont courts.

Plusieurs solutions sont, à cet effet, proposées. Nous allons envisager successivement celle de Randel, de Grau et de Gries. Lorsque le contenu d'un item équivaudra à celui donné dans la description logique des données, nous nous en tiendrons à le nommer; sans quoi, nous donnerons quelques brèves explications.

1. Dessin d'Article de RANDEL (2) - Whestone Compiler. Compilateur à 1 passage.

				[1]		[1]				[2]	[2]
ITEMS	NENTREE	NOM	TYPE U / CLASSE / GRAMM.	DECLARE	MEMOI-RE	SYLLA-BE	FORMEL	VALEUR	NDIM U / NPARAM	EXP.	LINE

(1) Bell estime à 20 % du Temps total de compilation, le temps passé à consulter la Table des Symboles. BELL, Quadratic Quotient, p. 107.

(2) RANDEL, Implementation, pp. 186 sqq.

[1] Soit l'exemple classique :

```
begin .....
      :
      :
      begin .....
            :
            :
            goto 1;
            :
            :
      end ;
 1:   ....
      :
      :
```

Naturellement, l'identificateur 1 (étiquette) est utilisé avant d'être déclaré. Cette information sera notée dans l'item DECLARE et l'endroit du Programme-Objet simultanément produit est mémorisé dans l'item SYLLABE. Lorsque viendra la déclaration de 1, la valeur de l'item DECLARE sera modifiée en conséquence. Il appert qu'à la fin du processus de compilation, tous les identificateurs doivent avoir été déclarés.
En fait, la structure d'accès imaginée par Randel généralise le procédé, mais il serait hors de notre propos d'en discuter plus longuement.

[2] Zones pour la récupération des erreurs.

2. Dessin d'Article de GRAU (1). Compilateur à 3 passages. Le premier passage construit essentiellement la Table des Symboles.

							[1]	[2]	
ITEMS	NOM	NENTREE	FORMEL ∪ VALEUR	TYPE	CLASSE GRAMM.	MEMOIRE	NBLOC	NIVTAB	SYLLABE

		[3]	[2]	[4]	[5]	[5]	[5]
ITEMS	NDIM ∪ NPARAM	LONGUEUR MEMOIRE	MAXNIVTAB	REFERENCE	OUTPUT	INPUT	ERREUR

(1) GRAU, Translation, pp. 132 - 135.

[1] En fait NPROC, car la compilation s'effectue en prenant en considération le niveau d'imbrication des procédures.

[2] Essentiellement des items qui permettent une gestion optimale de la Place-Mémoire dans l'utilisation des Tableaux.

[3] Cet item contiendra, au troisième passage, la longueur dans le Programme-Objet d'un corps de procédure.

[4] REFERENCE sert essentiellement, dans les cas des Tableaux, à indiquer l'adresse du vecteur d'adressage.

[5] Les trois dernières positions permettent de mémoriser les erreurs.

3. Dessin d'Article de GRIES (1) Compilateur ALCOR - 4 passages - implémenté sur une machine IBM 7090.

Chaque article a une longueur de 2 mots/mémoire (2 x 36 bits)

			[1]	[2]		
MOT 1	0 5	6	7 8	9 13	14 20	21 35
ITEMS	TYPE U CLASSE GRAMM.	FORMEL	DECLARE	NBLOC	NDIM U NPARAM	MEMOIRE

[1] Cfr. le problème des étiquettes traité pour le dessin d'article de RANDEL, note 1.

[2] Cfr. Dessin d'article de GRAU, note 1.

			[1]	[2]
MOT 2	0 2	3 17	18 20	21 35
ITEMS	VALEUR	NENTREE	-	BOUCLE

[1] Non mentionné.

[2] Position utilisée dans la procédure d'optimisation des boucles et de recouvrement des erreurs.

(1) GRIES, Compiler Construction, pp. 233 - 234.

2.3. Variables structurées

2.3.1. Définitions

Deux langages de programmation, communément utilisés de nos jours, PL/1 et COBOL, offrent à l'utilisateur la possibilité de définir ses données, en employant des variables structurées. Néanmoins, seule une structure arborescente statique y est admise. C'est pourquoi, il nous paraît naturel de rappeler, d'abord, la définition d'un arbre, de montrer ensuite la correspondance existant entre les termes COBOL (ou PL/1) et les notions impliquées par la définition de l'arbre.

2.3.1.1. Définition d'un arbre

Nous définirons "formellement un arbre comme un ensemble fini T de un ou plusieurs sommets tels que :

a) il y a un sommet particulier appelé racine de l'arbre;

b) les sommets restants (à l'exception de la racine) sont partitionnés en $m \geq 0$ ensembles $T_1 \ldots . T_m$ disjoints, et chacun de ces ensembles à son tour est un arbre. Les arbres $T_1 \ldots . T_m$ sont dits les sous-arbres (branches) de la racine [....] .

Il résulte de la définition que chaque sommet de l'arbre est la racine d'un sous-arbre contenu dans l'arbre tout entier. Le nombre de sous-arbres d'un sommet quelconque est dit le degré [sortant] de ce sommet. Un sommet de degré [sortant] 0 est un sommet terminal (feuille). Le niveau d'un sommet par rapport à T est défini en disant que la racine a le niveau 1 et les autres sommets ont un niveau de 1 supérieur à celui du sous-arbre de racine T_j qui les contient." (1).

---

(1) KNUTH, Art of Programming, I, p. 305.

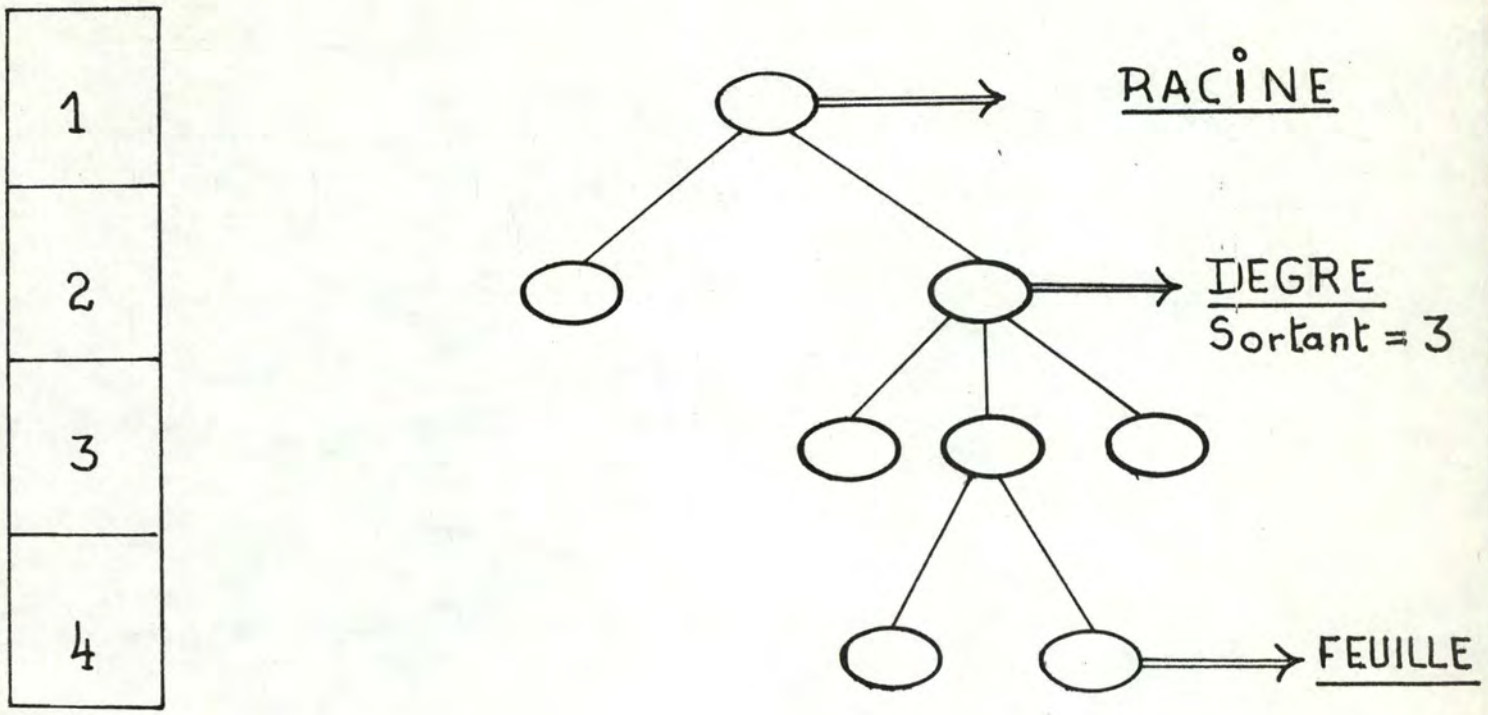


Fig. 2.3.1.1. Exemple d'arbre

2.3.1.2. Termes COBOL et Structure d'ARBRE

Etablissons, par un schéma, un parallèle entre les notions que nous venons de définir et celles dont use traditionnellement la littérature de COBOL ou PL/1.

THEORIE		COBOL (PL/1)
A R B R E	⟹	STRUCTURE
SOUS-ARBRE	⟹	GROUPE
FEUILLE	⟹	CONSTITUANT (élémentaire)

Fig. 2.3.1.2. (a) Lignes de Correspondance

La notion de niveau est maintenue telle quelle.

Ensuite, puisque notre propos est de manipuler des données à structure d'arbre, il convient que nous donnions à chacun des sommets,

un NOM qui est représentatif de VALEURS.

Habituellement, "on dira que la valeur d'un Groupe est la concaténation de ses constituants élémentaires" (1).

Ainsi, par exemple, la structure d'arbre de la Fig. 2.3.1.1. pourra prendre en COBOL l'aspect suivant :

```
O 1  A
   O 2  B  PI  C   I S   X...
   O 2  C
      O 3  D  PI C   I S   X...
      O 3  E
           O 4  G  P I C   I S   X...
           O 4  H  P I C   I S   X...
      O 3  F  PI C   I S   X...
```

Fig. 2.3.1.2. (b) Description de Données COBOL

2.3.1.3. Points d'entrée et référence

Lorsque, dans un traitement informatique, un utilisateur désire prendre en charge une donnée, il faut, comme on dit, qu'il la référence. En l'occurrence, la seule méthode générale pour désigner sans ambiguïté un constituant (2) consiste à préciser, par les noms des sommets, le chemin qui conduit du point d'entrée (Racine de l'arbre) à ce constituant.

Soit à référencer G dans la description de données Fig. 2.3.1.2.(b)

. COBOL : G OF E OF C OF A

. PL/1 : A.C.E. G.

On convient, cependant, de ne préciser que les sommets susceptibles de lever toute ambiguïté sur l'itinéraire à suivre. En l'occurrence G suffirait.

---

(1) CABANNES, Notes inédites, 2.7 .

(2) ... à la condition que le programmeur n'ait pas commis une faute de syntaxe !

2.3.2. Position du Problème

2.3.2.1. Objectif

Nous voulons, simplement, introduire dans la Table des Symboles les informations qui permettront à la fois de représenter la structure d'arbre sous-jacente à la description de la donnée (notons que cette information est proprement SEMANTIQUE), et de vérifierla correction de toute référence à un constituant - élémentaire ou non - de cette donnée.

2.3.2.2. Nos matériaux

Outre les définitions que nous venons d'établir, nous disposons des règles qui régissent la description de données hiérarchisées (1).

1) Chaque nom est précédé d'un entier positif, appelé son numéro de niveau;

2) Le nom de la structure est le premier composant dans la description d'une structure; son numéro de niveau doit être inférieur aux numéros de niveau de tous les autres constituants (élémentaires ou non);

3) Chaque nom de groupe est suivi par ses constituants. Les constituants doivent tous avoir le même numéro de niveau qui doit être plus grand que le numéro de niveau du nom de groupe;

4) Aucune règle n'impose l'unicité absolue des différentes valeurs des sommets; "simplement, deux constituants d'un même groupe et au même niveau ne peuvent porter la même dénomination". (2)
Mais la remarque est de bon sens !

2.3.2.3. Les solutions

Nous allons passer en revue les deux principales solutions décrites à ce jour. La première, que nous appellerons non-déterministe est due à Gries, la seconde, que par opposition, nous appellerons déterministe, a été proposée par Gates.

---

(1) GRIES, Compiler Construction, p. 236

(2) CABANNES, Notes inédites, 2.11.

2.3.3. Solution non-déterministe

2.3.3.1. Principe général

La définition même d'un arbre suggère une méthode quasi naturelle de le représenter à l'intérieur d'un ordinateur. En effet, à condition de toujours le parcourir de la racine vers les feuilles, il suffit d'associer à chaque sommet un nombre de pointeurs (1) équivalents au degré de ce sommet.

Ainsi, par exemple, voici une structure arborescente et sa représentation attendue.

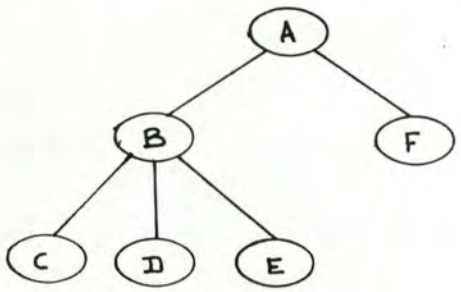


Fig. 2.3.3.1. (a) Structure arborescente

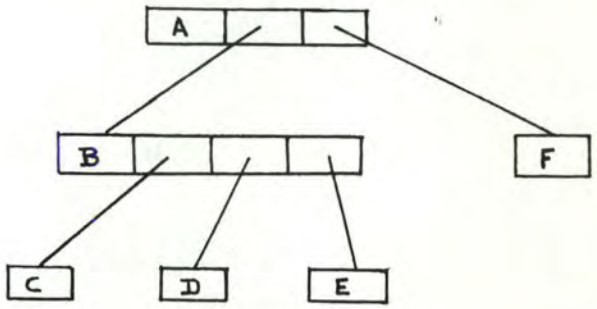


Fig. 2.3.3.1. (b) Représentation attendue

Mais, si, comme il semble normal, nous attribuons une entrée de la Table des Symboles à chaque constituant d'une structure, cette configuration présente l'inconvénient de rendre obligé l'emploi d'enregistrements à longueur logiquement variable : le nombre de pointeurs à chaque noeud est, vu de la machine, tout à fait aléatoire. C'est pourquoi Gries (2) préférant, d'une part, traiter des enregistrements de longueur fixe, et, d'autre part, vérifier la validité d'une référence au départ de la feuille de l'arbre (ou du constituant de niveau le plus élevé) a imaginé l'aménagement suivant. Outre ses items usuels (3), chaque entrée de la take est dotée de

---

(1) Il semble ressortir de l'article de GATES, Simple Technique, que cette méthode a été utilisée dans le compilateur COBOL, version 3.0, de Control Data. (Internal reference Specifications 64/65/6600, COBOL Compiler, version 3.0, Control Data Corporat., 1967)

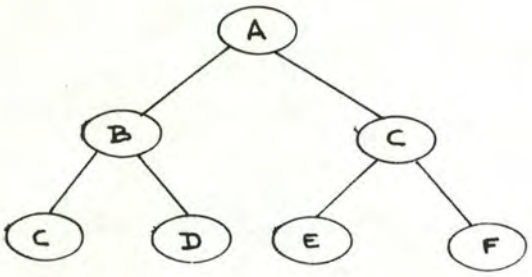
(2) GRIES, Compiler Construction, pp. 238-239.

(3) Cfr. 2.2.

quatre nouveaux items:

1) l'item PERE dont la valeur indique le numéro d'entrée dans la Table du père du constituant consulté; cette valeur = 0 si le constituant n'a pas de père (RACINE de l'arbre).
2) l'item FILS dont la valeur indique le numéro d'entrée dans la Table du premier FILS (le plus à gauche dans l'arbre) du constituant consulté; sa valeur = 0 si le constituant n'a pas de fils (FEUILLE).
3) l'item FRERE dont la valeur indique le numéro d'entrée dans la Table du frère suivant du constituant consulté (constituant de même niveau et immédiatement à sa droite dans l'arbre); sa valeur = 0, si le constituant n'a pas de frère. (1)
4) l'item SAME lie entre eux les constituants de même nom (cfr règle 4 de Description des Structures, p.26)

Un bref exemple fera mieux saisir ces notions. Soit la structure suivante sous sa forme arborescente :



1) A n'a pas de PERE; son premier fils est B et A n'a pas de frère (Unicité de la racine) !
2) B a pour PERE A , pour premier FILS C et pour premier FRERE suivant C ...

Fig. 2.3.3.1. (c) Structure sous sa forme arborescente

A supposer maintenant que nous voulions ranger ces renseignements dans la Table des Symboles, le résultat attendu devra présenter l'image suivante :

---

(1) Knuth note, non sans humour, que "certains auteurs préfèrent les désignations féminines 'mère, fille, soeur" au lieu de "père, fils, frère"; mais, pour bien des raisons, les termes masculins paraissent plus professionnels". Et de poursuivre sur sa lancée en écrivant que "d'autres auteurs encore préférant promouvoir l'égalité des sexes, utilisent les mots neutres de "parents, descendant, cousin" ! KNUTH, Art of Programming, I, pp. 307 - 309.

NUM. D'ENTREE (1)	N O M	S A M E	P E R E	F I L S	F R E R E
1	A	0	0	2	0
2	B	0	1	3	5
3	C	0	2	0	4
4	D	0	2	0	0
5	C	3	1	6	0
6	E	0	5	0	7
7	F	0	5	0	0

Fig. 2.3.3.1. (d) Résultat attendu du rangement dans une Table de la structure arborescente

Fig. 2.3.3.1. (c)

Nous allons montrer que l'algorithme qui suit parvient à ce résultat.

2.3.3.2. Algorithme de construction de la Table (2)

Nous supposerons qu'outre la Table des Symboles proprement dite dont la configuration serait celle de la Fig. 2.3.3.1. (d), nous disposons d'un vecteur IDEN, où les identificateurs sont rangés dans l'ordre de lecture du Programme-Source, et d'un vecteur NIVEAU donnant pour chaque identificateur son numéro de niveau.

La variable entière n donne la longueur logique de ces vecteurs (nombre d'identificateurs). Nous présumerons, en plus, l'existence d'une routine ADDENTRY à 2 paramètres P et Nom (3), dont le propos est d'entrer dans la Table des Symboles l'identificateur NOM,

(1) Nous les avons choisis délibérément séquentiels et en ordre croissant pour éviter de confondre STRUCTURE SEMANTIQUE et STRUCTURE D'ACCES (ordre des numéros d'entrée et manière de les retrouver).

(2) GRIES, Compiler Construction, p. 238 .... Ecrit en une espèce de PL/1.

(3) Son utilité apparaîtra plus tard.

d'initialiser tous ses composants à 0 sauf SAME, s'il existe déjà dans la Table un constituant portant le nom de l'identificateur à entrer.

Enfin, nous utiliserons pour chaque entrée une zone temporaire LEV, où sera chargé le contenu du vecteur NIVEAU relatif à l'identificateur entré.

P R O G R A M M E	C O M M E N T A I R E S (1)
BEGIN POINTER PO, Q,P ;	PO contiendra le numéro d'entrée dans la Table du nom de la structure (racine de l'arbre); Q contiendra le numéro d'entrée dans la Table de l'identificateur courant; P contiendra le numéro d'entrée dans la Table du père ou d'un frère éventuel de Q.
ADDENTRY (PO, IDEN [1]); P: = PO; PO.LEV : = NIVEAU [1] ;	Entrée du nom de la structure; PO est initialisé.
FOR i : = 2 STEP 1 UNTIL n DO BEGIN	Entrée successive de tous les éléments du vecteur IDEN.
BEGIN ADDENTRY (Q, IDEN [i]); Q.LEV : = NIVEAU [i] ;	
TEST : CASE SIGN (P.LEV - Q.LEV)+2 OF BEGIN	L'identificateur correspondant à l'entrée Q est-il le frère, le fils de l'identificateur correspondant à l'entrée ou bien ne sont-ils pas directement parents ?
BEGIN IF P.FILS ≠ 0 THEN STOP;	Première hypothèse : FILS. Si l'Item FILS n'est pas vide, il y a erreur puisque, par construction, nous ne prenons en compte que le premier FILS.
Q.PERE = P; P.FILS: = Q END	Sinon, il convient d'écrire l'information concernant la relation PERE - FILS trouvée.

(1) Nos commentaires sont plus détaillés que ceux donnés par Gries.

```
        BEGIN   Q.PERE : = P.PERE;
                P.FRERE: = Q
        END;
```

Deuxième hypothèse : FRERE. Ceci implique qu'ils ont le même PERE, et l'identificateur correspondant à l'entrée Q est le frère, immédiatement à droite dans l'arbre de l'identificateur dont le numéro d'entrée est P. Notons ces deux informations.

```
        BEGIN   P: = P.PERE;
                IF    P = O THEN ERROR;
```

Troisième hypothèse : Non directement parents.

Si l'identificateur correspondant à l'entrée P n'a pas de PERE, il y a erreur car seul, le nom de structure n'a pas de PERE. Dès lors, le seul autre moyen d'arriver à cette troisième hypothèse serait qu'un élément de la structure ait un numéro de niveau inférieur à celui du nom de la structure. Ce qui est interdit par la règle de description n° 2.

```
                GOTO  TEST;
```

Sinon, on cherchera à savoir si l'identificateur correspondant à l'entrée Q est lié au PERE de l'identificateur correspondant à l'entrée P.

```
        END;
END;
```

Fin de l'instruction CASE

```
        P: = Q;
```

Car la méthode vise à toujours se demander d'abord les relations unissant un identificateur à celui qui le précède dans la description. Il faut donc sauvegarder le numéro d'entrée de ce précédent.

END → Fin de l'Instruction FOR

IF PO. FRERE ≠ O THEN ERROR; END;	Test final visant à voir si la structure n'a réellement qu'une racine (dont le numéro d'entrée est conservé dans PO; sans quoi, bien entendu, la description proposée est erronée. (1)

Voici, à titre d'exemple, comment la structure décrite Fig. 2.3.3.1. (c) serait entrée dans la table.

1. Matériaux

a) Le SCANNER a introduit dans les vecteurs IDEN et NIVEAU les identificateurs et leur numéro de niveau. Le contenu de ces vecteurs est donc le suivant :

I D E N	A	B	C	D	C	E	F
N I V E A U	1	2	3	3	2	3	3

b) La longueur de ces vecteurs (n) = 7;

c) La Table des Symboles est supposée vide.

Dès lors, la construction se fera selon les étapes suivantes.

2. Construction

P A S 1

N° ENTREE	NOM	SAME	PERE	FILS	FRERE	LEV
1	A	O	O	O	O	1

Sortie : PO = 1; P = 1

P A S 2; i := 2; CASE (1) OF ...

N° ENTREE	NOM	SAME	PERE	FILS	FRERE	LEV
1	A	O	O	O	O	1
2	B	O	1	O	O	2

Sortie PO = 1; P = 2

(1) Correction d'une coquille dans l'algorithme de Gries. Il est bien sûr qu'il ne s'agit pas de vérifier, comme le laisserait entendre cette faute (PO.SON), Si la racine a un Fils (ce qu'elle a nécessairement) mais si elle a un FRERE (ce qu'elle ne peut avoir !). Gries suggère d'ailleurs la correction dans son propre commentaire ...

P A S 3 ; i := 3; CASE 1 OF ...

N° ENTREE	NOM	SAME	PERE	FILS	FRERE	LEV
1	A	O	O	2	O	1
2	B	O	1	3	O	2
3	C	O	2	O	O	3

Sortie : PO = 1, P = 3

P A S 4 ; i := 4; CASE 2 OF ...

N° ENTREE	NOM	SAME	PERE	FILS	FRERE	LEV
1	A	O	O	2	O	1
2	B	O	1	3	O	2
3	C	O	2	O	4	3
4	D	O	2	O	O	3

Sortie : PO = 1, P = 4

P A S 5; i := 5; CASE (3) OF...suivi de CASE (2) OF (GOTO TEST)

N° ENTREE	NOM	SAME	PERE	FILS	FRERE	LEV
1	A	O	O	2	O	1
2	B	O	1	3	5	2
3	C	O	2	O	4	3
4	D	O	2	O	O	3
5	C	3	1	O	O	2

Sortie : PO = 1; P = 5

P A S 6; i := 6; CASE (1) OF ...

N° ENTREE	NOM	SAME	PERE	FILS	FRERE	LEV
1	A	O	O	2	O	1
2	B	O	1	3	5	2
3	C	O	2	O	4	3
4	D	O	2	O	O	3
5	C	3	1	6	O	2
6	E	O	5	O	O	3

Sortie : PO = 1, P = 6

P A S 7; i := 7; CASE (2) OF ...

N° ENTREE	NOM	SAME	PERE	FILS	FRERE	LEV
1	A	O	O	2	O	1
2	B	O	1	3	5	2
3	C	O	2	O	4	3
4	D	O	2	O	O	3
5	C	3	1	6	O	2
6	E	O	5	O	7	3
7	F	O	5	O	O	3

CONCLUSIONS

1. Comme n = 7, sortie de la boucle FOR.
2. Test sur PO.FRERE; il est = 0; dès lors, l'écriture de la structure est correcte.
3. Le champ LEV peut être récupéré pour un usage ultérieur.

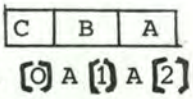
Sortie : PO = 1, P = 7.

2.3.3.3. Algorithme de Recherche d'une Référence

Il nous faut, à présent, construire un algorithme qui nous permettra de vérifier, à partir de la Table ainsi construite, la validité d'une référence à un constituant d'une variable structurée.

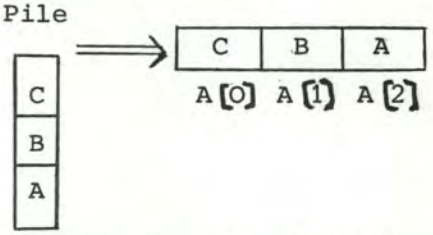
A cet effet, nous présupposerons qu'outre la Table des Symboles, nous disposons d'un STRING ARRAY A, dont la borne inférieure est 0. Nous y rangerons la référence à identifier, de telle manière que l'élément A [0] de ce tableau contienne le nom associé au sommet de l'arbre le plus éloigné de la Racine, c'est-à-dire l'ordre adopté par COBOL.

Ex: C OF A OF B ⟹



Dès lors, en PL/1, il faudra d'abord réorganiser la référence en l'introduisant au préalable dans une pile et en recopiant ensuite, à partir du sommet, le contenu de la pile dans le STRING ARRAY.

Ex: A.B.C. ⟹ Pile ⟹



Enfin, la variable entière n indique le nombre d'éléments du STRING ARRAY A pour une référence donnée.

P R O G R A M M E (1)	C O M M E N T A I R E S
BEGIN POINTER P,Q,S; INTEGER i; P:= "adresse d'une première entrée de nom (A [0])"; Q: = 0	Le pointeur P contient l'adresse d'une première entrée de la Table des Symboles, dont le nom = le contenu de l'élément A [0]. En réalité, cette terminologie nous paraît ambiguë. En effet, en

(1) GRIES, Compiler Construction, p. 239. Pour le langage utilisé, cf. Algorithme précédent.

ok	raison de la stratégie adoptée pour relier les différentes entrées de noms identiques (pointeur SAME), il vaudrait mieux dire "adresse de la dernière entrée chronologique de la Table dont le nom = A [O]".
WHILE P ≠ O DO	Le pointeur P va, en fait, contenir le cas échéant, les adresses successives de toutes les entrées de la Table dont le nom = CONTENU (A [O]).
BEGIN S: = P	S est un pointeur temporaire.
FOR i: = 1 STEP 1 UNTIL n DO	Le principe consiste à remonter la
BEGIN LOOP : S: = S. PERE	chaîne des PERES et à comparer chaque élément à l'élément courant du vecteur A.
IF S = 0 THEN GOTO TRYNEWP;	Si la racine de l'arbre a été rencontrée avant que la référence ait été épuisée, il ne s'agit donc pas de l'entrée dont l'adresse était dans le pointeur P. Aussi va-t-on tenter un nouvel essai (TRYNEWP) (1).
IF S.NOM ≠ A [1] THEN GOTO LOOP	Ce test est obligatoire car il n'est pas nécessaire que le chemin conduisant de la racine de l'arbre à une feuille soit complètement indiqué (cfr. p. 25)
END;	
IF Q ≠ O THEN ERROR;	Cette instruction sera exécutée si, et seulement si, il existe un chemin dans l'arbre correspondant à la référence proposée. Dès lors, si au moment de prendre en charge cette instruction, Q n'a plus sa valeur initiale O, cela signifie

(1) De là le côté NON DETERMINISTE DE LA SOLUTION : on essaie tous les candidats possibles jusqu'à ce qu'on en ait trouvé un et un seul !

	que l'algorithme a déjà produit un autre chemin correspondant à la même référence. Celle-ci est donc ambiguë et l'erreur est signalée.
Q: = P ;	On mémorise la première entrée qui satisfait à la référence proposée.
TRYNEWP : P: = P.SAME END;	Une entrée a échoué; l'essai est renouvelé avec une autre entrée de même nom, si du moins il en reste une (Test du WHILE).
IF Q = 0 THEN ERROR END;	Nous avons, de notre chef, ajouté cette instruction; mais elle nous semble indispensable dans la mesure où une erreur de référence intervient si la référence est impossible (de là notre test).

2.3.3.4. Conclusion

En fait, cette solution, essentiellement pragmatique, permet d'une part une construction aisée de la Table mais se trouve incapable, d'autre part, de déterminer la validité d'une référence du Programme-Source, sans parcourir tous les chemins possibles.

2.3.4. Solution déterministe

2.3.4.1. Principe général

La dernière constatation de notre conclusion amène Gates (1) à constater que le temps perdu à parcourir tous les chemins possibles pour chaque référence risque d'être grand, voire énorme, si les références sont nombreuses. Dès lors, ne serait-il pas plus efficace de construire expressément la Table en vue de la vérification ultérieure de la validité des références ?

---

(1) GATES, Simple Technique.

A cet effet, considérons d'une part la structure de données suivante, sommairement décrite en PL/1 :

```
1 A
 2 B
  3 C
  3 D
 2 C
  3 D
1 B
 2 A
```

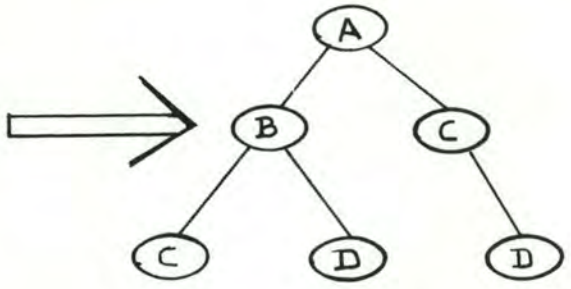


Fig. 2.3.4.1. (a)

Fig. 2.3.4.1. (b)

Considérons d'autre part les langages élémentaires :

- le langage vide $\emptyset$
- le langage $X = \{x\}$, c'est-à-dire le langage qui se réduit à la seule chaîne x.

Ces langages, selon la terminologie de Chomsky, sont évidemment réguliers. De même qu'est régulier le langage $\tilde{x} = \{\emptyset \cup X\}$, car "est régulier tout langage défini par une expression construite à partir des opérateurs réunion et concaténation $[\ldots]$ et de langages élémentaires" (1), c'est-à-dire par une expression régulière (2).

Mais alors, il existe un langage régulier défini par l'expression régulière :

$[\text{EXPR } 1]$ $A \cup (\tilde{B}\,(\tilde{C} \cup \tilde{D}) \cup \tilde{C}\,\tilde{D}) \cup B\,(\tilde{C} \cup \tilde{D}) \cup C\tilde{D} \cup C \cup D \cup B\tilde{A} \cup A$

Remarquons que cette expression décrit autrement la structure Fig. 2.3.4.1. (a), n'ajoutant aucune information supplémentaire; en d'autres termes, toutes les références possibles et légales sur cette structure sont de simples occurrences de l'expression régulière. Nous sommes donc, à présent, à même d'appliquer la théorie des automates finis. En effet, le langage régulier, défini par une expression régulière, est engendré par une grammaire régulière.

---

(1) LEROY, Cours, II, p.29.

(2) MINSKY, Computation, p. 72.

Or, "étant donné une grammaire régulière, on peut toujours construire un automate fini NON DETERMINISTE engendré par cette grammaire" (1).

Cet automate, en l'occurrence, sera construit par la procédure suivante :

PAS 1 : A partir d'un état initial S, chaque état est spécifié par la plus longue chaîne de qualifieurs possibles (le plus long chemin de la Racine de l'arbre à un sommet déterminé, en mentionnant tous les sommets rencontrés).
Les interconnections représentant ces plus longues chaînes de caractères sont introduites dans la Table de Transitions d'Etats.

Appliqué à la Fig. 2.3.4.1. (a), ce pas donne le résultat :

Qualifieurs / Etats	A	B	C	D
S	A	B	--	--
A	--	A.B	A.C	--
B	B.A	--	--	--
A.B	--	--	A.B.C	A.B.D.
A.C	--	--	--	A.C.D.
B.A	--	--	--	--
A.B.C	--	--	--	--
A.B.D	--	--	--	--
A.C.D	--	--	--	--

Fig. 2.3.4.1. (c)

Dans la terminologie de Moore (2), les qualifieurs sont les Symboles d'Entrée, la Matrice Fig. 2.3.4.1. (c), est l'expression sous forme tabulaire d'un sous-ensemble de valeurs de la fonction de Transition φ à 2 arguments (un symbole d'entrée et un état de l'automate) qui a pour valeur un nouvel état de l'automate.

Ex. : φ (C, A.B) = A.B.C.

Enfin, la fonction de réponse ψ vaut 1 (Etat acceptant) pour tous les états de la colonne états, sauf S .

---

(1) LEROY, Cours, p. II, 18

(2) LEROY, Cours, p. II, 4

PAS 2 : Pour obtenir la fermeture de l'automate (de l'expression régulière), il importe que tous les états possibles apparaissent, avec leurs transitions.
Dès lors, puisque l'état S permet d'atteindre l'état A, on copiera la ligne A dans la ligne S et la procédure sera répétée jusqu'à épuisement des états.
L'automate final non déterministe se présente ainsi :

Qualif. / Etats	A	B	C	D
S	A,B.A	B,A.B	A.C, A.B.C	A.B.D,A.C.D
A	--	A.B	A.C,A.B.C	A.B.D,A.C.D
B	B.A	--	--	--
A.B	--	--	A.B.C	A.B.D
A.C	--	--	--	AC.D
B.A	--	--	--	--
A.B.C	--	--	--	--
A.B.D	--	--	--	--
A.C.D	--	--	--	--

Fig. 2.3.4.1. (d) Automate final non déterministe

Cet automate, cependant, n'apporte aucune efficience nouvelle à la procédure classique : par sa nature non déterministe, l'énumération de toutes les solutions sera requise, lorsqu'on voudra vérifier la validité d'une référence. Mais la théorie des automates finis nous permet de passer à un stade déterministe, c'est-à-dire à un stade où,en un seul passage dans l'automate, nous pourrons décréter qu'une référence est illégale, permise ou ambiguë. En effet, "soit L un langage accepté par un automate fini non déterministe, alors il existe un automate fini déterministe qui accepte L." (1).

(1) HOPCROFT, Formal Languages, p. 31 (On pourra y lire la preuve).

Pour construire cet automate déterministe équivalent, "il suffit de considérer les ensembles d'états de ce dernier. On définit l'ensemble initial comme $\{S\}$.

Puis, l'ensemble successeur d'un ensemble d'états pour un symbole d'entrée x est défini comme la réunion des ensembles successeurs de ces états pour x tels qu'ils sont définis par la fonction de Transition" (1).

Dès lors, l'automate déterministe équivalent à l'automate non déterministe Fig. 2.3.4.1. (d) se présente de la sorte :

Combinaison	Qualif. / Etat	A	B	C	D
S	1	2	3	4	5
A, B.A	2	--	6	4	5
B, A.B	3	7	--	8	9
A.C,A.B.C	4	--	--	--	10
A.B.D,A.C.D	5	--	--	--	--
A.B	6	--	--	8	9
B.A	7	--	--	--	--
A.B.C	8	--	--	--	--
A.B.D	9	--	--	--	--
A.C.D	10	--	--	--	--

Fig. 2.3.4.1. (e) Automate déterministe (2)

Cet automate est utilisé au départ de l'état S et en lisant les qualifieurs (Symboles d'entrée) de la gauche vers la droite.

Une référence sera dite illégale, si la liste des transitions possibles est épuisée avant la liste des qualifieurs. Par exemple, la référence B.A.B.

Une référence sera dite permise, si, lorsque la liste des qualifieurs est épuisée, l'automate se trouve dans un état qui représente un état unique de l'automate non déterministe. Exemple de références permises: A.B, B.A , A.B.C, B.C , B.D, C.D.

(1) LEROY, Cours, II p.16.
(2) GATES, Simple Technique, p. 563.

Une référence sera dite ambiguë, si l'état final de l'automate déterministe représente 2 ou plusieurs états de l'automate non déterministe. Ainsi A.D, A.B, A.C.

2.3.4.2. Algorithme de construction de la Table et du premier pas de l'automate non déterministe.

L'implémentation de cette méthode se fera en deux temps. Dans une première étape, nous allons construire la Table des Symboles et le premier pas de l'automate non déterministe (cfr. Fig.2.3.4.1 (c)). Ensuite, simultanément, nous procéderons à la fermeture de l'automate non déterministe et nous le rendrons déterministe. Nous supposerons que nous avons à notre disposition une Table de Symboles divisée en deux parties : NOM et REFERENCE. Chaque élément de NOM contiendra la représentation d'un nom de constituant et un pointeur vers la première référence possible de ce nom dans la zone REFERENCE; chaque élément de REFERENCE comportera les Items suivants : le nom de la référence, la valeur qui lui est associée et un pointeur vers une éventuelle référence suivante de ce nom.

Soit la structure :

	NOM	
	Nom	Pointeur
1 A		
2 B	A	→
3 C	B	→
3 A	C	→
2 C		

Fig. 2.3.4.2. (a)

Représentation attendue

	REFERENCE		
	Référence	Valeur(1)	Pointeur
	A	'valeur'	
	A.B	'valeur'	0
	A.B.C	'valeur'	
	A.B.A	'valeur'	0
	A.C	'valeur'	0

Fig. 2.3.4.2. (b)

(1) Répétons que dans la zone 'valeur', nous retrouvons tous les renseignements traditionnellement rangés dans une Table de Symboles. Cfr. Section 2.2.

Néanmoins, pour la facilité de l'exposé, nous symboliserons chaque élément des zones NOM ou REFERENCE sous formes de boîtes numérotées selon leur ordre d'entrée (chaque boîte porte naturellement un numéro différent), tandis que les pointeurs seront remplacés par un arc orienté.

Dès lors, la structure décrite Fig. 2.3.4.2. (a) sera représentée de la sorte :

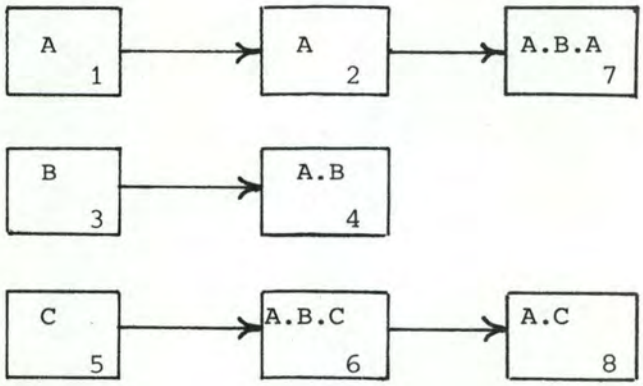


Fig. 2.3.4.2. (c)

Ainsi, dorénavant, lorsque nous désignerons une référence, nous le ferons par son numéro de boîte.

En outre, nous présumerons que nous disposons d'une pile appelée ETAT-COURANT, initialisée à 0 et d'une matrice (MATRICE TRANSITION) à 3 colonnes, successivement nommées ETAT-COUR, SYMBENT, TRANSIT. A la fin de la procédure, cette matrice se veut l'image de la matrice Fig. 2.3.4.1. (c).

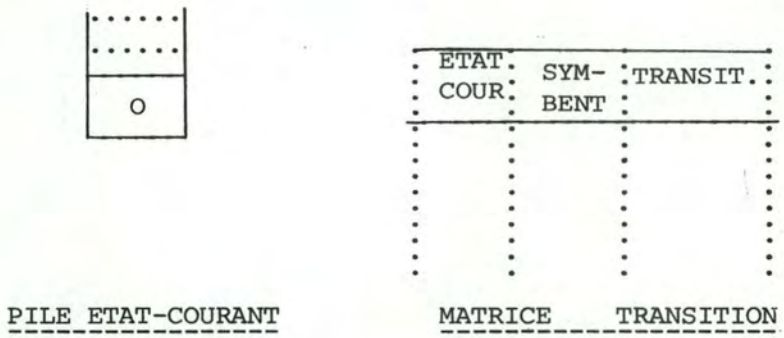


Fig. 2.3.4.2. (d)

Dès lors, l'algorithme se déroule LOGIQUEMENT (1) de la manière suivante :

PAS	A L G O R I T H M E (2)
1	Prendre le nom suivant dans la structure (initialement, le premier).
2	Consulter la Table des Symboles (partie NOM); si le nom s'y trouve rangé, aller en 3; sinon, ajouter ce nom dans le vecteur NOM.
3	Ajouter une nouvelle occurrence de ce nom, avec la valeur appropriée, dans la zone REFERENCE.
4	Ajouter, dans la matrice TRANSITION, une transition (sommet de Etat-Courant, Symbole d'Entrée, Occurrence du Nom particulier). Comme prévu, on manipulera les numéros de boîtes de préférence aux noms ou références (occurrences du nom).
5	Si la description de la structure est terminée, fin de la procédure; sinon, comparer le numéro de niveau de l'identificateur courant à celui de l'identificateur suivant.
6	Si ce numéro est inférieur, ajouter la nouvelle occurrence (plutôt le numéro de sa boîte) au sommet de la pile ETAT-COURANT et aller en 1.
7	Si ce numéro est identique, aller en 1.
8	Si ce numéro est supérieur, enlever du sommet de la pile ETAT-COURANT un nombre d'éléments égal à la différence entre les deux numéros de niveau et aller en 1.

Pour que le processus soit bien clair, nous allons illustrer l'évolution des différentes zones de mémoire au fur et à mesure que sont

---

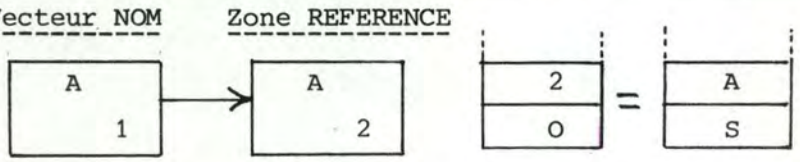
(1) Nous resterons ainsi au niveau où l'auteur a situé sa solution. Le lecteur ne perdra pas de vue cependant qu'il s'agit de construire 3 TABLES en Mémoire !

(2) Cet algorithme est, en substance, le même que celui proposé par Gates . Nous y avons introduit, néanmoins, quelques éclaircissements qui paraissaient indispensables.

lus les symboles de la structure décrite Fig. 2.3.4.1. (a) p.

I. SYMBOLE D'ENTREE 'A' Pas suivis : 1, 2, 3, 4, 5, 6

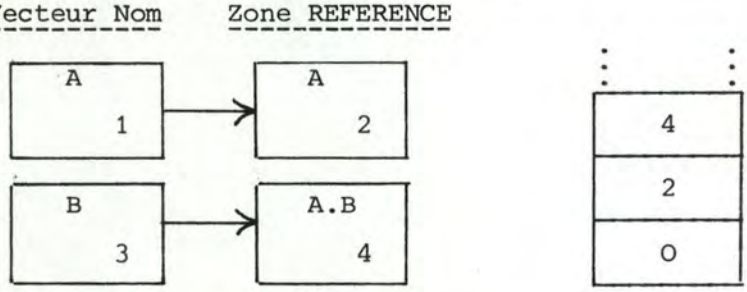
TABLE DES SYMBOLES PILE ETAT-COURANT MATRICE TRANSITION



ETAT-COUR	SYMBENT	TRANSIT
O	1	2
S	A	A

II. SYMBOLE D'ENTREE 'B' Pas suivis : 1, 2, 3, 4, 5, 6

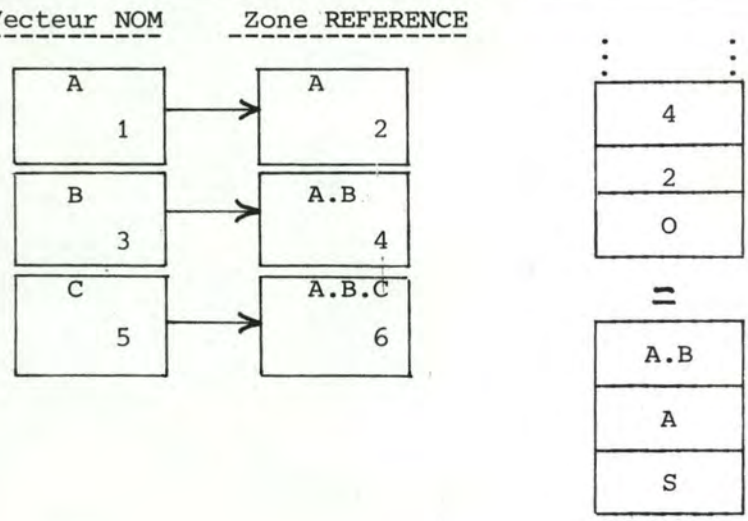
TABLE DES SYMBOLES PILE ETAT-COURANT MATRICE TRANSITION



ETAT-COUR	SYMBENT	TRANSIT
O	1	2
2	3	4

III. SYMBOLE D'ENTREE 'C' Pas suivis : 1, 2, 3, 4, 5, 7

TABLE DES SYMBOLES PILE ETAT-COURANT MATRICE TRANSITION



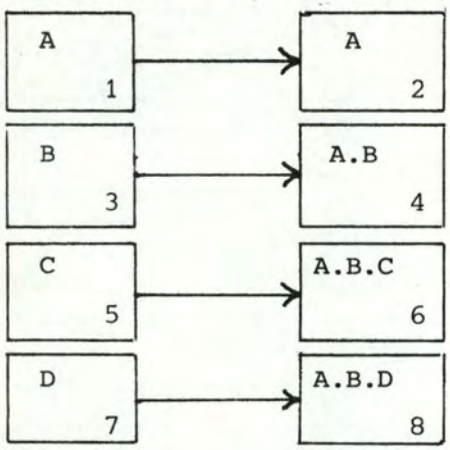
ETAT-COUR	SYMBENT	TRANSIT
O	1	2
2	3	4
4	5	6

S	A	A
A	B	A.B
A.B	C	A.B.C

IV. SYMBOLE D'ENTREE 'D' Pas suivis : 1,2, 3, 4, 5, 8

TABLE DES SYMBOLES PILE ETAT-COURANT MATRICE TRANSITION

Vecteur NOM Zone REFERENCE



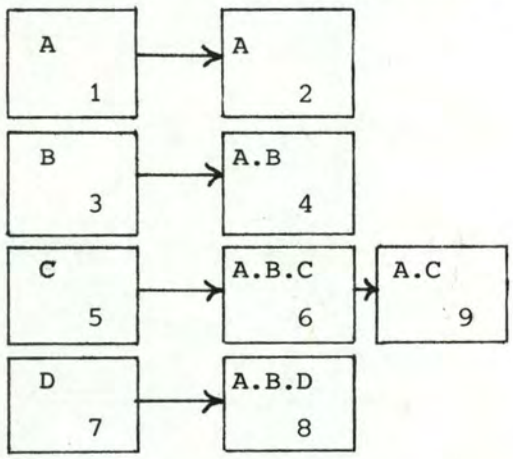
2
0

ETAT-COUR	SYMBENT	TRANSIT
0	1	2
2	3	4
4	5	6
4	7	8

V. SYMBOLE D'ENTREE 'C' Pas suivis : 1, 2, 3, 4, 5, 6

TABLE DES SYMBOLES PILE ETAT-COURANT MATRICE TRANSITION

Vecteur NOM Zone REFERENCE

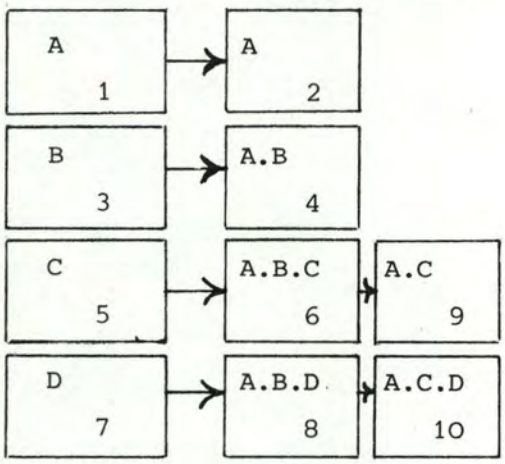


9
2
0

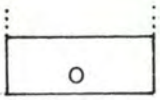
ETAT-COUR	SYMBENT	TRANSIT
0	1	2
2	3	4
4	5	6
4	7	8
2	5	9

VI. SYMBOLE D'ENTREE 'D' Pas suivis : 1, 2, 3, 4, 5, 8

TABLE DES SYMBOLES



PILE ETAT-COURANT

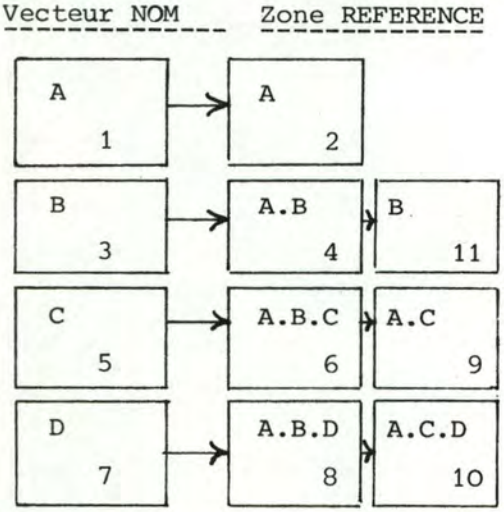


MATRICE TRANSITION

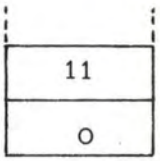
ETAT-COUR	SYMBENT	TRANSIT
0	1	2
2	3	4
4	5	6
4	7	8
2	5	9
9	7	10

VII. SYMBOLE D'ENTREE 'B' Pas suivis : 1, 2, 3, 4, 5, 6

TABLE DES SYMBOLES



PILE ETAT-COURANT



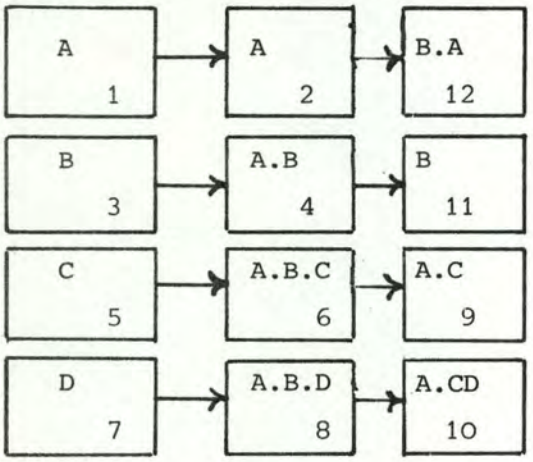
MATRICE TRANSITION

ETAT-COUR	SYMBENT	TRANSIT
0	1	2
2	3	4
4	5	6
4	7	8
2	5	9
9	7	10
0	3	11

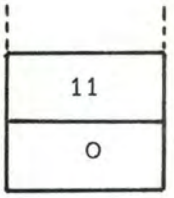
VIII. SYMBOLE D'ENTREE 'A' Pas suivis : 1, 2, 3, 4, 5

TABLE DES SYMBOLES
Etat final

Vecteur NOM | Zone REFERENCE



PILE ETAT-COURANT



MATRICE DE TRANSITION
Etat final

ETAT-COUR	SYMBENT	TRANSIT
0	1	2
2	3	4
4	5	6
4	7	8
2	5	9
9	7	10
0	3	11
11	1	12

2.3.4.3. Algorithme de construction de la fermeture de l'automate non déterministe, et de l'automate déterministe

A cet effet, nous supposerons que nous disposons :

1) de la Table des Symboles, telle qu'elle vient d'être construite à l'étape précédente;

2) de l'état final de la MATRICE TRANSITION
Avant toutes choses, nous la trierons, en ordre descendant des clés, sur l'item ETAT-COUR. Pour distinguer la matrice ainsi ordonnée de la précédente MATRICE TRANSITION, nous l'appellerons MATRICE INITIALE. Son image apparaît Fig. 2.3.4.3. (a).

ETAT-COUR	SYMBENT	TRANSIT
11	1	12
9	7	10
4	5	6
4	7	8
2	3	4
2	5	9
0	1	2
0	3	11

Fig. 2.3.4.3. (a) MATRICE INITIALE

3) d'une zone de mémoire extensible, que nous désignerons par ZONE DE TRAVAIL. Initialement, elle comporte deux colonnes ; INPUT et TRANS (Fig. 2.3.4.3. (b);

ZONE DE TRAVAIL

INPUT	TRANS

Fig. 2.3.4.3. (b) ZONE DE TRAVAIL (état initial)

4) d'une case-mémoire du nom d'ETAT-COURANT et primitivement vide;

5) d'une matrice à 3 colonnes (Fig. 2.3.4.3. (c), appelée MATRICE FINALE. A la fin de la procédure, elle représentera l'automate déterministe.

ETAT	SYMBOLE	SUIVANT

(1)

Fig. 2.3.4.3. (c) MATRICE FINALE (état initial)

(1) Nous avons changé le nom des colonnes par simple commodité (souci de clarté). Il est évident, en effet, que ces items désignent la même réalité que les items ETAT-COUR, SYMBENT et TRANSIT de la MATRICE INITIALE.

L'algorithme de construction se déroule dès lors, comme suit :

PAS	ALGORITHME
1	Pour l'état suivant (colonne ETAT-COUR) de la MATRICE INITIALE, transférer le contenu des zones SYMBENT et TRANSIT, respectivement dans les colonnes INPUT et TRANS de la ZONE DE TRAVAIL; sauver le contenu de ETAT-COUR dans la case ETAT-COURANT.
2	Pour chaque état apparaissant dans la colonne TRANS de la ZONE DE TRAVAIL, ajouter, dans ZONE de TRAVAIL, toutes les transitions de cet état pour les divers symboles d'entrée. Par l'ordre adopté dans l'assignation des adresses (construction du pas 1 de l'automate non déterministe), ces transitions apparaissent déjà dans la MATRICE FINALE ou n'existent pas.
3	Trier, en ordre ascendant, le contenu de ZONE DE TRAVAIL sur l'item INPUT. Conserver une seule transition par état et symbole d'entrée, ce qui revient à combiner en un seul ensemble les différentes transitions possibles (colonne TRANS de ZONE DE TRAVAIL) pour un symbole d'entrée déterminé (colonne INPUT de ZONE DE TRAVAIL).
4	Ajouter le contenu de ETAT-COURANT dans la colonne ETAT de la MATRICE FINALE et le contenu de ZONE DE TRAVAIL, respectivement dans les colonnes SYMBOLE et SUIVANT de la MATRICE FINALE. Vider ZONE DE TRAVAIL et ETAT-COURANT.
5	Rendre l'automate déterministe. A cet effet, remplacer, par un état unique, les combinaisons d'états obtenues au pas 3. On procédera de la sorte : a) Générer un nombre unique pour cette combinaison. Ce nombre sera différent de ceux déjà attribués et référera à une zone mémoire où sera gardée trace de la combinaison. Sauver ce nombre dans la case ETAT-COURANT; b) Pour chacun des états de la combinaison remplacée, copier dans la ZONE DE TRAVAIL, le contenu de SYMBOLE et SUIVANT de la MATRICE FINALE. En effet, toujours en raison de lbrdre d'attri-

5 (suite)

bution des numéros de boîtes (voir pp.42.), les états de la combinaison se trouvent déjà, avec leurs transitions, dans la MATRICE FINALE.

c) Exécuter les Pas 3 et 4 de l'algorithme;

d) Répéter jusqu'à ce que toutes les combinaisons d'états soient ainsi résolues, et retourner en 1.

Proposons-nous, à titre d'exemple, d'examiner, pas après pas, le contenu des différentes zones de mémoire, lorsque sera pris en charge l'état 2 de la colonne ETAT-COUR de la MATRICE INITIALE. Avant d'entrer dans la procédure, la situation se présente de la sorte :

1) Table des Symboles

NOM	REFERENCE
A	A → B.A
B	A.B → B
C	A.B.C → A.C
D	A.B.D → A.C.D

2) MATRICE INITIALE

ETAT-COUR	SYMBENT	TRANSIT
11	1	12
9	7	10
4	5	6
4	7	8
2	3	4
2	5	9
0	1	2
0	3	11

Etat à prendre en charge

3) MATRICE FINALE

ETAT	SYMBOLE	SUIVANT
11	1	12
9	7	10
4	5	6
4	7	8

4) ZONE DE TRAVAIL

INPUT	TRANS
⋮	⋮ ⋮

5) Case ETAT-COURANT

PAS 1 (Sortie) : Copie dans ZONE DE TRAVAIL des transitions de la MATRICE INITIALE correspondant à l'état 2; mémorisation dans ETAT-COURANT de l'état analysé.

MATRICE FINALE

ETAT	SYMBOLE	SUIVANT
11	1	12
9	7	10
4	5	6
4	7	8

ZONE DE TRAVAIL

INPUT	TRANS
3	4
5	9

ETAT-COURANT

2

PAS 2 (Sortie) : Fermeture de l'état 2 dans ZONE DE TRAVAIL.

MATRICE FINALE

ETAT	SYMBOLE	SUIVANT
11	1	12
9	7	10
4	5	6
4	7	8

ZONE DE TRAVAIL

INPUT	TRANS	
3	4	
5	9	
5	6	} transitions de l'état 4
7	8	} transitions de l'état 4

ETAT-COURANT

2

PAS 3 (Sortie) : Tri, en ordre ascendant, de ZONE DE TRAVAIL sur l'Item INPUT et création d'une seule transition par état.

MATRICE FINALE

ETAT	SYMBOLE	SUIVANT
11	1	12
9	7	10
4	5	6
4	7	8

ZONE DE TRAVAIL

INPUT	TRANS
3	4
5	{6 U 9}
7	8

ETAT-COURANT

2

PAS 4 (Sortie) : Copie de ZONE DE TRAVAIL et de ETAT-COURANT dans la MATRICE FINALE

MATRICE FINALE

ETAT	SYMBOLE	SUIVANT
11	1	12
9	7	10
4	5	6
4	7	8
2	3	4
2	5	{6 U 9}
2	7	8

ZONE DE TRAVAIL

INPUT	TRANS
-	-

ETAT-COURANT

-

PAS 5 : Génération d'un état unique à la place de la combinaison d'états {9 U 6}

a) Génération d'un nombre unique (SORTIE)

MATRICE FINALE

ETAT	SYMBOLE	SUIVANT
11	1	12
9	7	10
4	5	6
4	7	8
2	3	4
2	5	13
2	7	8

Combinaison {6 U 9}

ZONE DE TRAVAIL

INPUT	TRANS
-	-

ETAT-COURANT

-

b) Copie dans ZONE DE TRAVAIL des transitions au départ de l'état 13; mémorisation de 13 dans ETAT-COURANT.

MATRICE FINALE

ETAT	SYMBOLE	SUIVANT
11	1	12
9	7	10
4	5	6
4	7	8
2	3	4
2	5	13
2	7	8

ZONE DE TRAVAIL

INPUT	TRANS
7	10

de l'état 9

ETAT-COURANT

13

c) Mise à jour de la MATRICE FINALE

Pas 4 : Tri de ZONE DE TRAVAIL. Le contenu des différentes zones reste identique.

Pas 5 : Recopie de ZONE DE TRAVAIL dans la MATRICE FINALE.

MATRICE FINALE

ETAT	SYMBOLE	SUIVANT
11	1	12
9	7	10
4	5	6
4	7	8
2	3	4
2	5	13
2	7	8
13	7	10

ZONE DE TRAVAIL

INPUT	TRANS
-	-

ETAT-COURANT

-

Combinaison {6 U 9}

Telle est donc la configuration de la MATRICE FINALE, lorsque l'algorithme a pris en charge l'état 2 de la MATRICE INITIALE. Si, pour être plus concret, nous remplaçons les numéros par les symboles qu'ils représentent, cette matrice s'écrira également :

ETAT	SYMBOLE	SUIVANT
B	A	B.A
A.C	D	A.C.D
A.B	C	A.B.C
A.B	D	A.B.D
A	B	A.B
A	C	{A.B.CUA.C}
A	D	A.B.D
A.B.C U A.C	D	A.C.D

Quant à l'automate déterministe, il aura, à la fin de la procédure, l'aspect suivant :

ETAT	SYMBOLE	SUIVANT	
11	1	12	
9	7	10	
4	5	6	
4	7	8	
2	3	4	
2	5	13	Combinaison des états 6 et 9
2	7	14	" " 8 et 10
13	7	10	
0	1	15	" " 2 et 12
0	3	16	" " 4 et 11
0	5	13	" " 6 et 13
0	7	14	" " 8 et 10
15	3	4	
15	5	13	
15	7	14	
16	1	12	
16	5	6	
16	7	8	

2.3.4.4. Recherche d'une référence

L'automate déterministe que nous venons de construire a, en fait, la forme d'une TABLE. Par conséquent, lorsque nous aurons à analyser une référence, nous procéderons à une recherche en Table selon une des techniques dont nous allons débattre longuement plus avant.

Au préalable, nous aurons pris soin de trier cette Table, en ordre ascendant, sur l'item ETAT. Il importe, en effet, que toute recherche parte de l'état O. Le résultat de ce travail apparaît Fig. 2.3.4.4.

	ETAT	SYMBOLE	SUIVANT	
1 ⟶	O	1	15	Combinaison 2 et 12
	O	3	16	Combinaison 4 et 11
	O	5	13	Combinaison 6 et 9
	O	7	14	Combinaison 8 et 10
	2	3	4	
	2	5	13	
	2	7	14	
3 ⟶	4	5	6	
	4	7	8	
	9	7	10	
	11	1	12	
	13	7	10	
2 ⟶	15	3	4	
	15	5	13	
	15	7	14	
	16	1	12	
	16	5	6	
	16	7	8	

Fig. 2.3.4.4. MATRICE FINALE triée

Le principe général est alors le suivant. Soit à vérifier la validité de la référence A.B.C. Partant de l'état O, au symbole d'entrée A (Boîte 1) est associé l'état suivant 15; de l'état 15, pour le symbole d'entrée B (Boîte 3), on passe à l'état 4; enfin de l'état 4, le symbole d'entrée C (Boîte 5) conduit à l'état 6 (état final). Les entrées de la matrice successivement visitées sont entourées sur la Fig. 2.3.4.4.

A ce stade là, trois hypothèses sont possibles :

a) La séquence des qualifieurs est épuisée et l'état final de l'automate ne représente pas une combinaison d'états (Ex: 4, 6, 8); la référence est dite correcte, et nous obtiendrons l'information qui lui est relative dans la zone REFERENCE de la Table des Symboles (cfr Fig. 2.3.4.2. (b)) à la boîte correspondant au numéro de l'état final de l'automate.

b) La séquence des qualifieurs n'est pas épuisée et l'automate se trouve dans un état sans transition. Supposons qu'il faille vérifier la validité de la référence A.B.A. L'automate passera successivement par les états O, 15, 4, où il ne rencontrera aucune transition pour le symbole A (n° de boîte 1). La référence est donc erronée.

c) La séquence des qualifieurs est épuisée et l'état final de l'automate correspond à 1 combinaison d'états (Dans notre exemple, les états 13, 14, 15, 16); la référence est alors dite ambiguë et l'erreur est signalée. Ainsi, la référence A.C aura suivi le parcours O, 15, 13 (état final).

2.3.4.5. Conclusion

Selon les auteurs, cette méthode semble donner de meilleurs résultats lorsque le nombre de références à rechercher est grand. (1)

---

(1) GATES, Simple Technique, p. 565.

Cette affirmation, quoique fort prudente déjà, nous apparaît encore sujette à caution. En effet :

1) il faudrait, pour qu'elle soit fondée, se livrer à des études statistiques approfondies, portant sur le temps de compilation car
   a) d'une part, il est indéniable qu'au regard de la solution non-déterministe, le temps de construction de la Table est accru dans de notables proportions (la logique du système telle que nous l'avons décrite ne donne qu'un piètre reflet de sa complexité réelle).
   Si bien que le gain total de temps escompté lors de la vérification des références d'un programme s'en trouve amenuisé d'autant !
   b) d'autre part, une consultation de Table n'est ni aussi "simple" (1), ni aussi déterministe qu'on voudrait le faire croire. Le lecteur se reportera, à ce propos, aux pp.63ssq de notre travail.
2) Les auteurs ont choisi l'exemple-limite où chaque identificateur apparaît plusieurs fois. Et ce choix s'avérait indispensable pour justifier la méthode déterministe. Cependant, cette situation est-elle, dans les programmes réels, si banale ? N'est-il pas naturel, au contraire, qu'un utilisateur cherche à individualiser ses identificateurs dans le dessein d'éviter, au maximum, des erreurs de logique ? Et même, un compilateur doit-il être conçu en fonction des esprits tortueux ?
   Encore une fois, seule une étude statistique apporterait une réponse détaillée et circonstanciée.

Aussi bien, nous paraît-il de loin préférable de nous en tenir à la solution de Gries, simple à implémenter et efficace dans des circonstances "normales".

---

(1) ... "simple look-up" dira Gates.

3. LA RECHERCHE DES INFORMATIONS DANS LA TABLE DES SYMBOLES

3.1. Introduction

3.1.1. Définition d'un problème de recherche

Tout problème de recherche, en informatique, se ramène au problème général suivant : "Comment ranger l'enregistrement d'une information dans une collection d'enregistrements (dans notre propos, la Table des Symboles) de sorte qu'on puisse le retrouver ?"

Sans autre précision, ce problème est insoluble. C'est pourquoi, toute solution suppose, au préalable, que soit donné un élément capable d'identifier univoquement un enregistrement : son adresse ou encore une clé.

Les algorithmes de recherche ont dès lors pour objectif, à partir d'une clé K et d'une collection d'enregistrements E :

1) de trouver l'enregistrement $E_k \in E$ qui a K pour clé, dans l'hypothèse où cet enregistrement appartient à la collection désignée;
2) d'insérer l'enregistrement E_k de clé K dans la collection E, s'il n'y appartient pas déjà. (1)

Plus formellement, quoique moins complètement, nous pourrions dire avec Hopgood (2), qu' "étant donné une clé K, une fonction M ($\approx$ algorithme de recherche) est définie de telle sorte que M(K) est une information dérivée" (plus concrètement, l'adresse de l'enregistrement E_k dans la collection E, soit qu'il s'y trouve, soit qu'il faille l'y insérer).

---

(1) KNUTH, Art of Programming, III, p. 389.

(2) HOPGOOD, Compiling Techniques, p. 16.

3.1.2. Méthodes de recherche par clé et modes logiques d'accès.

Il est permis de partitionner l'ensemble des méthodes de recherche par clé en deux grandes classes (1) :

1) les méthodes de recherche basées sur de simples comparaisons entre les clés;
2) les méthodes de recherche utilisant les propriétés digitales des clés.

A chacune de ces classes est associé, comme nous allons le montrer, un mode particulier d'accès logique, c'est-à-dire décidé par le programmeur.

3.1.2.1. Comparaison de clés et accès logique par itinéraire

Concrètement, la méthode de recherche basée sur de simples comparaisons de clé fonctionne de la sorte :

1) à partir d'une clé donnée K_D , d'une collection d'enregistrements E dont le premier enregistrement est E_o, on se pose la question de savoir si la clé K_O de cet enregistrement = K_D .
2) Si oui, la recherche est fructueuse;
3) Sinon, on se donne, selon des techniques que nous étudierons, un autre enregistrement. Et ainsi de proche en proche, jusqu'à ce que la recherche ait donné un résultat fructueux ou que la collection E à investiguer soit épuisée (Recherche infructueuse).

Si bien que la structure d'accès logique correspondant à cette méthode de recherche peut être représentée sous la forme d'un graphe dont chaque sommet constitue un enregistrement logique. Un arc orienté ira du sommet A au sommet B si et seulement si, l'enregistrement A permet d'accéder à l'enregistrement B.(2)

---

(1) KNUTH, Art of Programming, III, p. 389.

(2) CHERTON, Cours, V. 10.

Si lbn considère la Fig. 3.1.2.1., il est dès lors évident que pour accéder à l'enregistrement E_5, il faut logiquement suivre l'itinéraire "E_o, E_3, E_5" ou encore l'itinéraire "E_o, E_2, E_5".

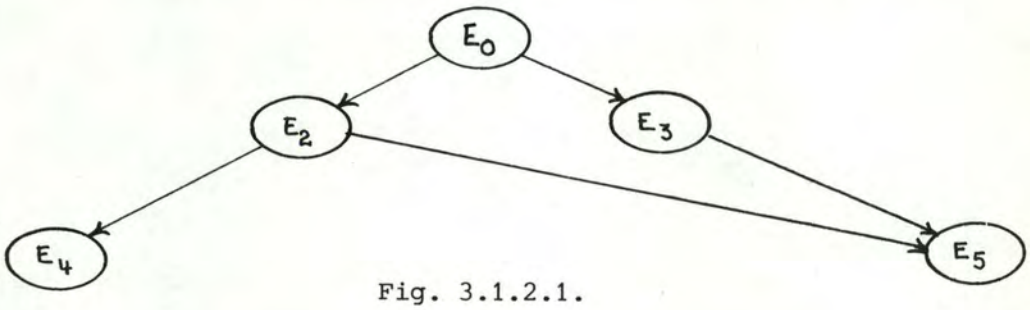


Fig. 3.1.2.1.

D'autre part, il n'est pas possible d'accéder à l'enregistrement E_3 à partir de l'enregistrement E_2.

3.1.2.2. Propriétés digitales des clés et accès logique calculé (1)

Cette méthode tire parti de la configuration interne d'une clé K (une suite de O et 1) et utilise cette information comme base de la recherche.

En fait, elle choisit la clé K comme argument d'une fonction F dont la valeur sera le numéro ou l'adresse de l'enregistrement E_k de clé K dans la collection E.

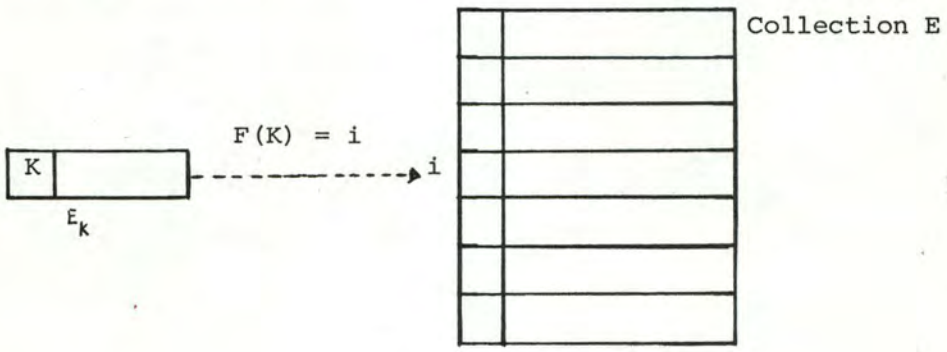


Fig. 3.1.2.2. Configuration générale d'une méthode d'accès logique calculé.

En d'autres mots, une fonction F associe à une clé K, l'entier i, qui est l'adresse logique (relative au début de la collection) ou physique de l'enregistrement de clé K dans la collection E.

---

(1) La terminologie est sur ce point mal fixée encore. Les Américains parlent, en général, de SCATTER STORAGE.

3.1.2.3. Conclusion

Ainsi s'éclaire le rôle véritable de l'algorithme de recherche : il ne vise à rien d'autre qu'à mettre en concordance la structure logique d'accès ("par itinéraire" ou "associatif") et la structure physique d'accès, du mode "par adresse" puisque nous travaillerons en Mémoire Centrale.

3.1.3. Plan du chapitre

Pour chacune des deux classes de méthodes de recherche, nous nous proposons :

1) de décrire brièvement les différentes techniques qui transforment le mode logique d'accès en mode physique;
2) de donner l'algorithme de recherche le plus précisément qu'il se peut. A ce stade, nous l'appellerons, sans que le contenu en soit modifié, l'algorithme d'implémentation.
3) d'établir pour chacune des techniques utilisées leurs performances. Ce qui postule, bien entendu, que nous nous soyons donné des critères de mesure. En fait, pour l'essentiel, deux mesures, toujours les mêmes, seront prises en compte :
   a) le temps que nous exprimerons, sous forme logique, en termes de nombre moyen d'accès nécessaires dans le cadre d'une recherche fructueuse (nous le désignerons par C_N, N représentant le nombre d'enregistrements présents dans la collection E) ou infructueuse (C'_N)
   La relation liant ce temps "logique" au temps "réel" nous paraît immédiate, sauf pour les méthodes HASH sans chaînage . (1)
   b) la place-mémoire que nous évaluerons à partir de la constatation triviale : "la place minimale occupée par N enregistrements de longueur M = N x M".

---

(1) Voir plus loin p. 101

3.2. Recherche par simples comparaisons de clés

Nous allons envisager successivement trois techniques bien connues et communément appelées : recherche séquentielle, recherche dichotomique (1), recherche par arbre binaire.

En fait, les deux dernières, surtout, sont d'application, dans le contexte des Tables de Symboles du Compilateur. Cependant, par souci de cohérence dans la démarche, nous exposerons d'abord - et succinctement - la première de ces méthodes.

3.2.1. Recherche séquentielle (Compilateur ALCOR - ALGOL 60 - GRIES)

3.2.1.1. Description générale

La méthode de recherche séquentielle repose sur l'a priori suivant : " si un enregistrement logique (sommet) A permet d'accéder à un enregistrement logique (sommet) B, c'est-à-dire s'il existe un arc orienté d'origine A et d'arrivée B, alors A et B sont contigus sur le support d'information.

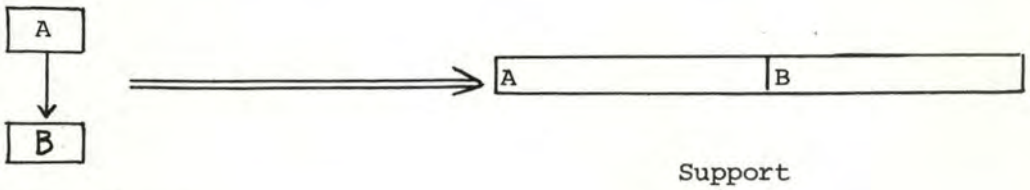


Fig. 3.2.1.1. (a)

Dès lors, la structure d'accès logique, inhérente à cette méthode, peut être représentée sous la forme d'un graphe élémentaire (Fig. 3.2.1.1. (b) où pour chaque sommet le degré entrant et le degré sortant sont tous deux égaux à 1, sauf pour deux sommets particuliers appelés premier et dernier pour lesquels ces degrés valent

---

(1) ou bien binaire, logarithmique ...

respectivement (0,1) et (1,0). (1)

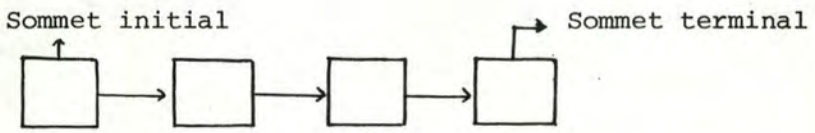


Fig. 3.2.1.1. (b)

Le principe de recherche sera donc le suivant. Supposons que nous voulions vérifier si l'enregistrement E_K de clé K appartient à la collection d'enregistrement E et, le cas échéant, que nous ayons à l'y insérer.

1) Nous comparerons d'abord la clé K à la clé du premier enregistrement de la collection E sur le support.
2) Si ces clés sont identiques, la recherche est fructueuse; sinon, nous comparerons la clé K à la clé de l'enregistrement suivant sur le support. Et ainsi jusqu'au moment où :
   a) nous aurons trouvé un enregistrement E_K de clé K;
   b) nous aurons épuisé la collection E auquel cas, nous écrirons l'enregistrement E_K de clé K à la suite du dernier enregistrement de la collection E.

Cette procédure est illustrée par cet exemple. Soit la collection E de 6 enregistrements (N = 6) et l'enregistrement E_K de clé K pour lequel nous devons opérer une recherche.

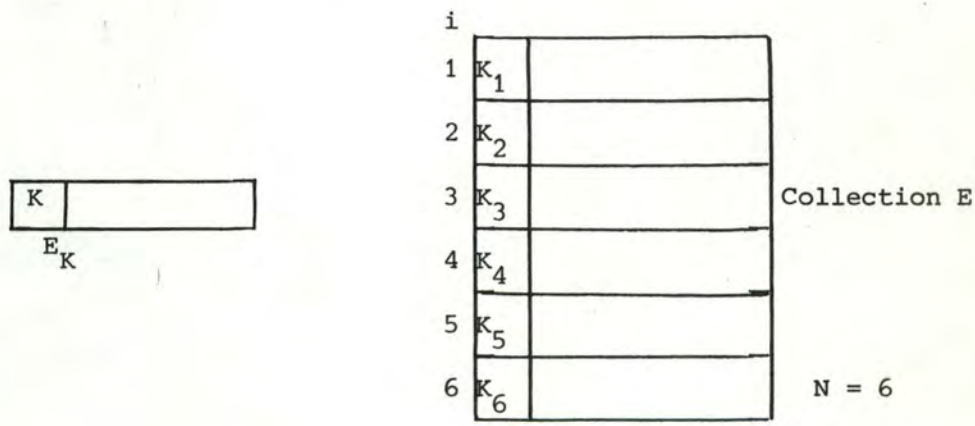


Fig. 3.2.1.1. (c) Configuration du problème

(1) CHERTON, Cours, V, 10.

Prenons les deux hypothèses possibles :

1) $K \in \{K_1, \ldots K_6\}$; par exemple $K = K_3$; alors la procédure se déroulera ainsi :

Test	Réponse	Action
$K = K_1$?	NON	
$K = K_2$?	NON	
$K = K_3$?	OUI	Recherche fructueuse

2) $K \notin \{K_1, \ldots K_6\}$; par exemple $K = K_7$; alors nous procéderons de la manière suivante :

Test	Réponse	Action
$K = K_1$?	Non	
⋮		
$K = K_6$?	Non	La collection E est épuisée. Nous écrirons donc l'enregistrement E_K à la suite de l'enregistrement de clé K_6 et la Table sera ainsi modifiée, Fig. 3.2.1.1. (d).

K_1	
K_2	
K_3	
K_4	
K_5	
K_6	
K	

Fig. 3.2.1.1.(d) Collection E après la recherche de K. (N = 7)

En pareille occurrence, un algorithme d'implémentation, non nécessairement le meilleur, apparaît à l'évidence. (1)

(1) KNUTH, Art of Programming, III, pp. 393-394.

3.2.1.2. Algorithme d'Implémentation

Etant donné une collection E d'enregistrements R_1, R_2, ...R_N rangés de manière contiguë en Mémoire Centrale, de clé respective K_1, K_2, ... K_N, de longueur fixe (1), alors pour un argument K donné et N (nombre d'enregistrements dans la Table) ≥ 1, l'algorithme se déroule de la sorte, si i désigne le numéro des entrées de la Table.

PAS	ALGORITHME	COMMENTAIRE
1	i := i + 1	Initialisation de la variable i; la recherche commence toujours au premier enregistrement de la collection E.
2	K : K_i ?	K est-il ou non égal à K_i ?
3	Si oui, la recherche est fructueuse; sinon, i := i + 1.	
4	i $\leqslant$ N ? Si oui, aller en 2;	A-t-on épuisé la collection d'enregistrement E ?
5	Appeler la procédure 'Recherche infructueuse' .	Ecrire l'enregistrement E_K de clé K à la suite du dernier enregistrement de la collection E.

(1) La condition n'est pas nécessaire; nous l'avons retenue pour la facilité de l'exposé.

3.2.1.3. Performances

a) Temps logique

Le nombre moyen d'accès pour une recherche fructueuse dans une Table de N enregistrements $\boxed{(C_N) = \frac{N + 1}{2}}$, tandis que le nombre moyen d'accès dans l'hypothèse d'une recherche infructueuse $\boxed{(C'_N) = N.}$

b) Place Mémoire

Il est clair que la place occupée par le fichier en Mémoire Centrale est minimale; en d'autres termes, la place requise par un fichier de N enregistrements logiques de longueur M, accédé selon un mode de recherche séquentielle, égale exactement N * M.

En conclusion, les performances "temps" sont si médiocres (par exemple, si N = 1000, il faudra 500 accès en moyenne pour une recherche fructueuse), que ce type de recherche est la plupart du temps abandonnée.

3.2.2. Recherche dichotomique

3.2.2.1. Description

Outre leur contiguïté sur le support, cette technique exige des enregistrements qu'ils soient rangés dans l'ordre croissant des clés. De cette propriété découle le principe de recherche suivant.
Soit à déterminer si un enregistrement E de clé K appartient à la Table des Symboles T. Dans un premier temps, nous comparerons la clé K de E_K à la clé de l'enregistrement dont l'adresse logique dans T = $\frac{N+1}{2}$, si N représente le nombre total d'enregistrements entrés dans T à ce moment là.

Selon la réponse, . la recherche aura été fructueuse ($K = K_{(N+1)/2}$

. la recherche sera poursuivie sur les entrées appartenant à l'intervalle $]{(N+1)/2} \quad N]$, si $K > K_{(N+1)/2}$

. la recherche sera poursuivie sur les entrées appartenant à l'intervalle $[1 \quad (N+1)/2 [$, dans l'hypothèse où $K < K_{(N+1)/2}$

Et ainsi, de proche en proche, jusqu'à ce que notre recherche se soit révélée fructueuse ou infructueuse. En ce cas, l'enregistrement E_K de clé K est donc à insérer dans la Table T, sous la contrainte initiale d'ordre croissant des clés. Ce qui implique que la Table soit triée, lors de toute nouvelle entrée.

En général, il paraît préférable - lorsque le problème le permet - de trier la Table, lorsque tous les identificateurs y ont été entrés.

Si donc, maintenant, nous désirons représenter, sous forme de graphe, la structure d'accès logique inhérente à cette méthode de recherche, nous obtenons un arbre bien équilibré comme celui de la Fig. 3.2.2.1.

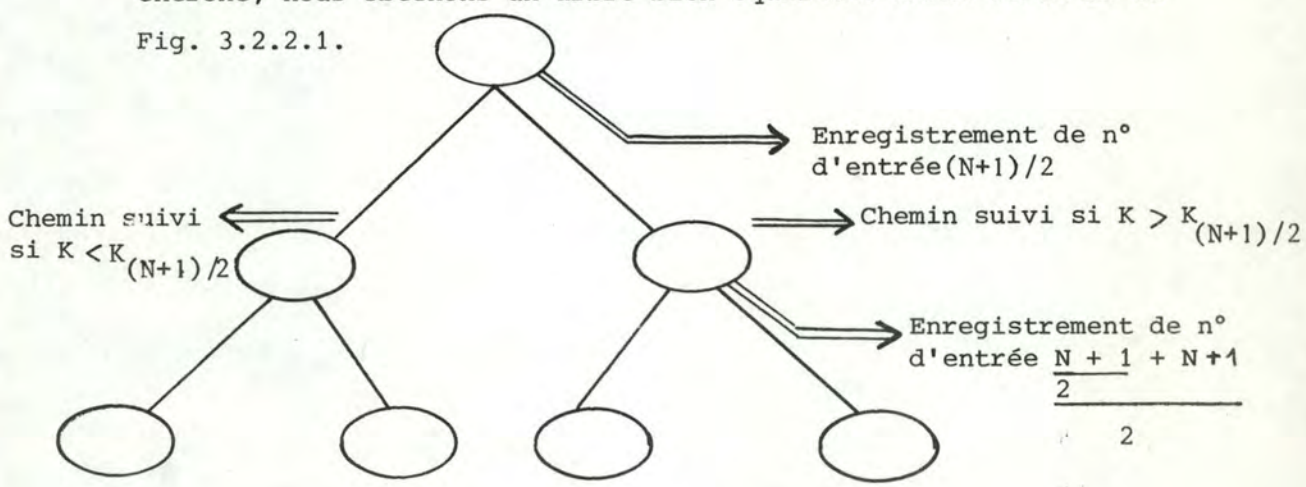


Fig. 3.2.2.1. Graphe de la structure d'accès inhérente à la recherche dichotomique.

Evidemment, si la procédure de tri s'effectue en même temps que la procédure d'entrée, les sommets du graphe changeront de valeur à chaque nouvelle insertion.

3.2.2.2. Algorithme d'Implémentation

Etant donné une Table d'enregistrements R_1 .... R_N de longueur fixe, rangés de manière contiguë en mémoire centrale et dont les clés sont dans l'ordre croissant $K_1 < K_2 .... < K_N$,alors pour un argument K donné et $N \geqslant 1$, l'algorithme se déroule ainsi, si la procédure d'insertion a lieu au fur et à mesure des entrées.

PAS	ALGORITHME (1)	COMMENTAIRES
1	l := 1; u := N;	Phase d'initialisation : les pointeurs l et u indiquent à tout moment les limites inférieure et supérieure de l'intervalle de recherche.
2	u < l ? Si oui, l'algorithme se termine sans succès. Sinon, i := (l + u)/ 2	Partant de l'hypothèse que si K est dans la Table, il satisfait à la relation $K_l \leqslant K \leqslant K_u$,nous testons d'abord si u < l. Appel de la procédure d'insertion. Nous calculons, en l'occurrence, le milieu de l'intervalle de recherche.
3	Comparons K à K_i ; Si $K < K_i$, aller en 4; Si $K > K_i$, aller en 5; Si $K = K_i$, la recherche est fructueuse.	L'ordre des clés permet, à ce stade-ci, d'affiner la comparaison, par rapport à la méthode de recherche séquentielle.
4	u := i - 1; aller en 2;	Réaménagement de la borne supérieure de l'intervalle.
5	l := i + 1; aller en 2 .	Réaménagement de la borne inférieure de l'intervalle.

(1) KNUTH, Art of Programming, III, p.407.

3.2.2.3. Performances

a) Temps

Le nombre moyen d'essais en cas de recherche fructueuse

$$\boxed{C_N \approx \log_2 N - 1;} \qquad (1)$$

le nombre moyen d'essais en cas de recherche infructueuse

$$\boxed{C'_N \approx \log_2 N + 1;} \qquad (1)$$

A quoi il faut ajouter le temps de "tri", c'est-à-dire, essentiellement le temps-machine nécessaire pour mouvoir d'une case tous les enregistrements dont la clé est supérieure à la clé de l'enregistrement inséré. (En temps logique, au mieux $N\log_2 N$ accès, selon Lee (2)).
Au total, cette méthode de recherche se révèle peu efficace dès que $N > 30$.

b) Place-mémoire

Naturellement la place requise demeure minimale.

3.2.3. Recherche par "arbre binaire" (compilateur FORTRAN G WATFOR)

3.2.3.1. Description

Un arbre binaire est, par définition, un arbre (3) dont chaque sommet est racine de deux sous-arbres au plus; lorsqu'un seul de ces sous-arbres est présent, nous sommes amené à distinguer le sous-arbre de gauche et le sous-arbre de droite (4).

Fig. 3.2.3.1. (a)

En fait, l'algorithme de recherche dichotomique construit implicitement un arbre binaire : l'ordre croissant des clés garantit en effet une construction univoque de l'arbre, dès que le premier intervalle de recherche est connu.

Cependant, la contrainte de tri, inhérente à celle d'ordre total des clés, alourdit, de façon quasi intolérable, les performances de temps. Aussi voudrait-on s'en affranchir. En résulteraient :

1) la possibilité d'entrer les enregistrements dans la Table selon leur ordre d'arrivée, sans que l'hypothèse de contiguïté en Mémoire Centrale soit encore requise;
2) la nécessité de construire explicitement l'arbre puisqu'évidemment, la contrainte d'ordre étant supprimée, le parcours n'est plus univoquement déterminé dès que sont connues les bornes de l'intervalle de recherche;
3) la nécessité de décider, sur l'arbre, un ordre de parcours basé sur un certain ordre des clés. En général, la solution retenue suppose que chaque enregistrement E_K de clé K représente un sommet de l'arbre; l'ensemble (éventuellement vide) des enregistrements (sommets) du sous-arbre de gauche ont une clé $<$ K (selon l'ordre alphabétique, par exemple), tandis que l'ensemble (éventuellement vide) des enregistrements du sous-arbre de droite ont une clé $>$ K.

   Ainsi, imaginons que nous voulions construire l'arbre de parcours correspondant à des enregistrements de clé A, B, C, D, E, F.

   a) Ces clés sont arrivées dans l'ordre C, D, A, E, B, F.

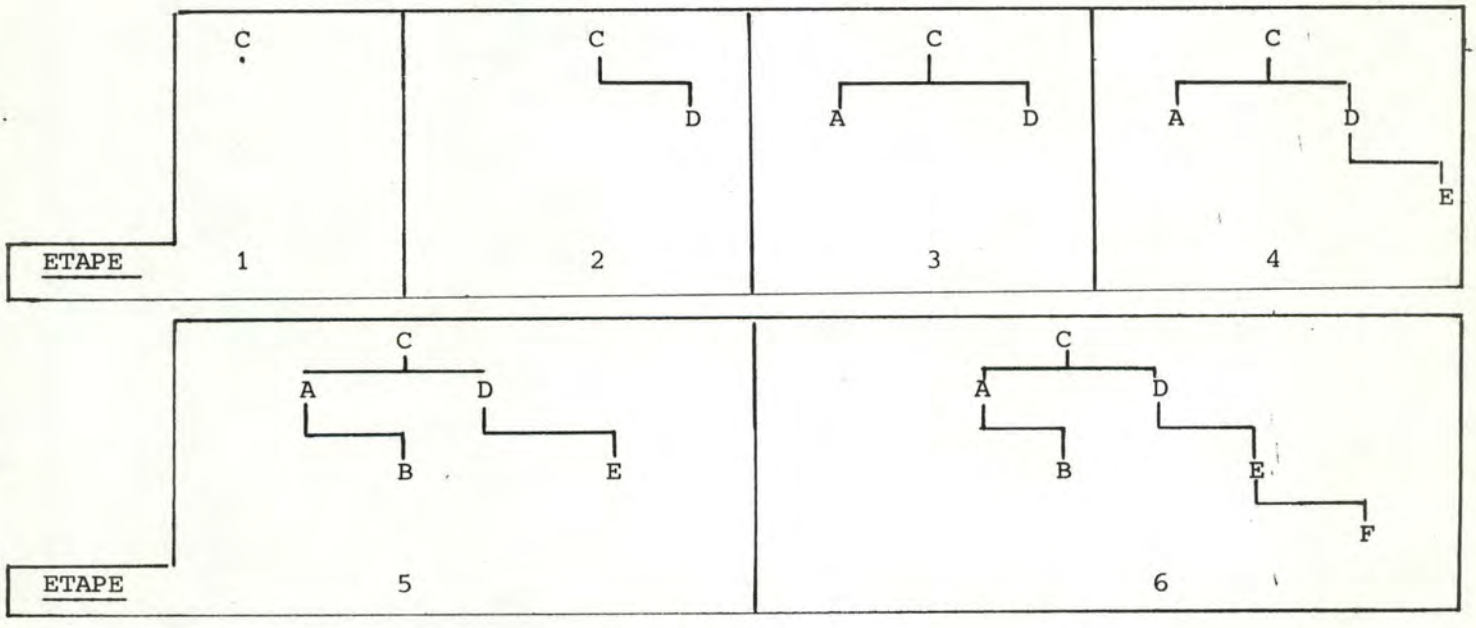


Fig. 3.2.3.1. (b)

b) Ces clés sont arrivées dans l'ordre A, B, C, D, E, F.

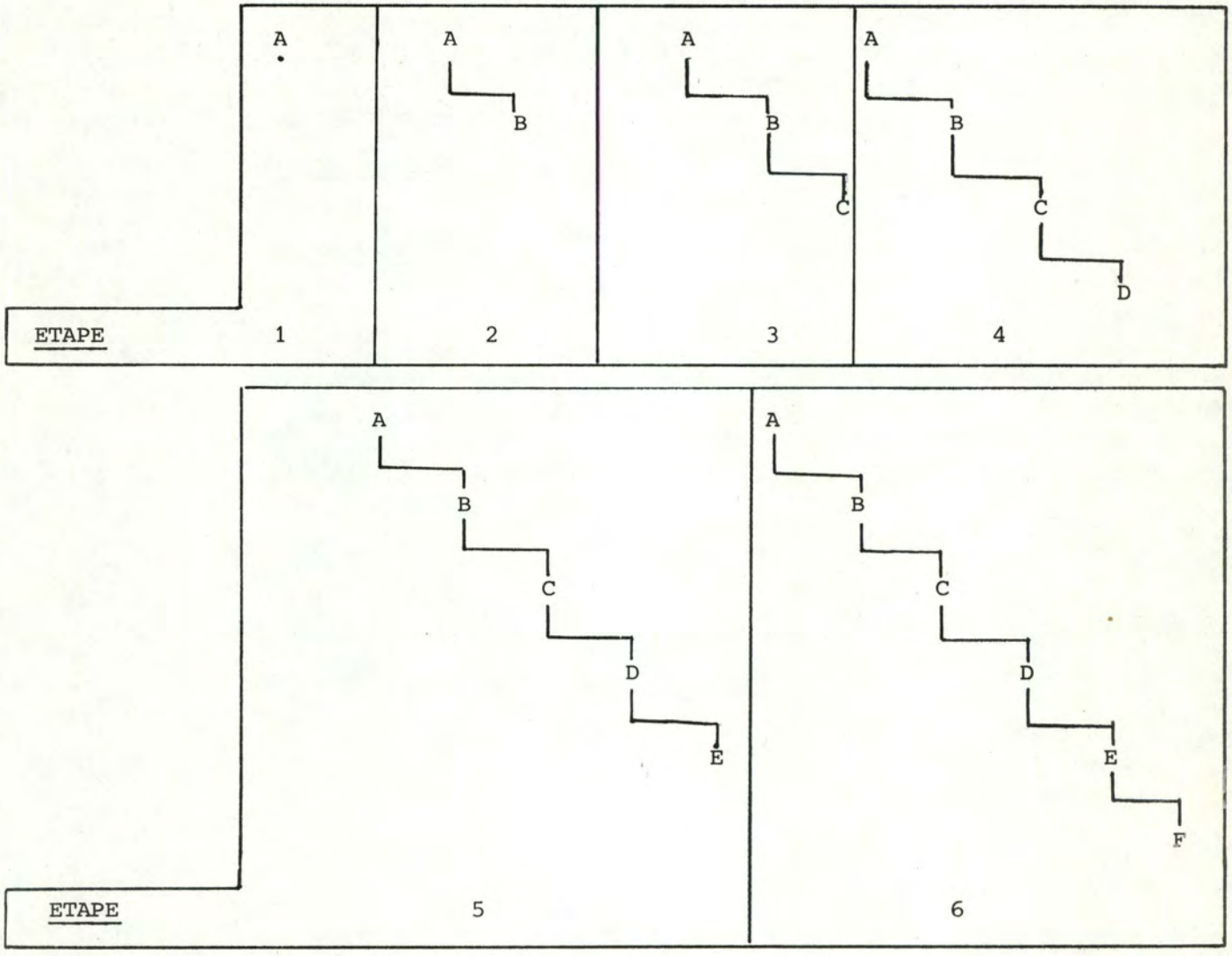


Fig. 3.2.3.1. (c)

Nous reviendrons plus avant sur les conséquences apparentes de l'ordre d'arrivée (1).

3.2.3.2. Algorithme d'Implémentation

En toute généralité, la représentation interne d'un itinéraire suppose l'usage de pointeurs qui, pour chaque enregistrement E, indiquent le ou les enregistrements accessibles à partir de E.

(1) Cfr. p. 75

A titre d'exemple, voici la représentation interne conforme à l'arbre décrit Fig. 3.2.3.1. (b).

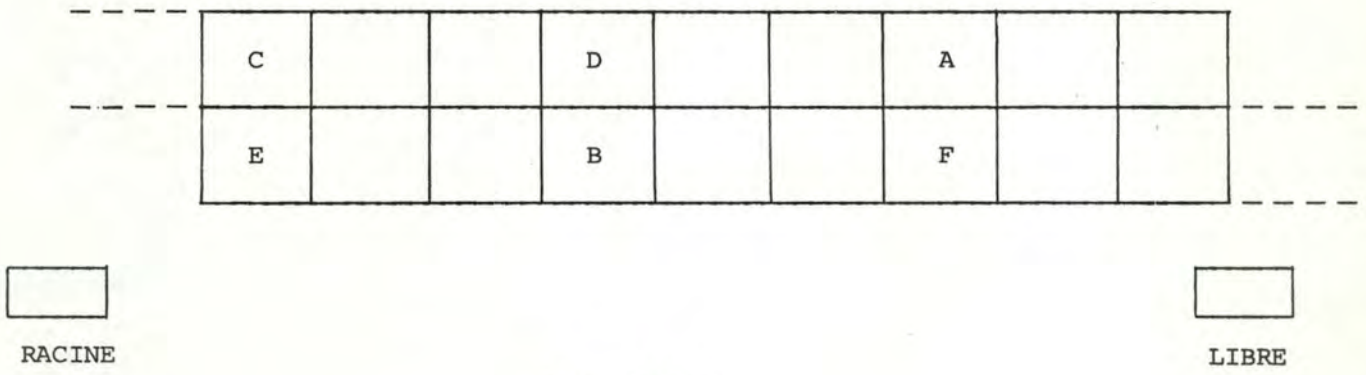


Fig. 3.2.3.2.

L'algorithme suivant construit une Table ainsi structurée, à la condition que deux hypothèses soient satisfaites :

1) les enregistrements contiennent au moins 3 zones de longueur fixe ou variable, peu importe :
   - CLE où sera rangé l'identificateur;
   - POINT GAUCHE, qui pointe vers le sous-arbre de gauche et POINT DROIT vers le sous-arbre de droite.
2) la variable RACINE pointe vers la racine de l'arbre;
   la variable LIBRE pointe vers la première place libre en Mémoire Centrale;
3) L'arbre n'est pas vide.

Soit à "rechercher" l'enregistrement E_K de clé K.

PAS	ALGORITHME (1)	COMMENTAIRES
1	P := RACINE	Initialisation du pointeur P dont le rôle sera de progresser dans l'arbre.
2	Comparer K à CLE [P]	Comparer la clé de l'enregistrement E_K à la clé de l'enregistrement d'adresse P.

(1) KNUTH, Art of Programming, III, p. 424.

	Si K < CLE [P], aller en 3; Si K > CLE [P], aller en 4; Sinon, la recherche est fructueuse.	Parcours du sous-arbre de gauche. Parcours du sous-arbre de droite.
3	POINT GAUCHE [P] ≠ 0 ? Si oui, P := POINT GAUCHE [P] et aller en 2; Sinon, aller en 5 ...	P va contenir l'adresse du prochain sommet de l'arbre à interroger. .. et déclencher la procédure d'insertion.
4	POINT DROIT [P] ≠ 0 ? Si oui, P := POINT DROIT [P] et aller en 2; Sinon, aller en 5.	Remarques identiques à celles du pas 3.
5	Q := LIBRE CLE [Q] := K POINT GAUCHE [Q]:= 0 POINT DROIT [Q] := 0 Si K<CLE[P], POINT GAUCHE[P] := Q; Si K>CLE[P], POINT DROIT[P] := Q. LIBRE := LIBRE + Longueur de E_K	La procédure d'insertion se déroule en 4 phases : 1) le pointeur Q contient l'adresse où il convient d'insérer le nouvel enregistrement. 2) Entrée de l'enregistrement E_K de clé K à l'adresse Q. 3) Mise à jour de la structure d'accès. 4) Mise à jour du pointeur LIBRE.

3.2.3.3. Performances

a) Temps logique

L'exemple que nous avons développé Fig. 3.2.3.1. (b) et 3.2.3.1. (c) pourrait donner à penser que le Temps logique d'accès dépend étroitement de l'ordre d'entrée des identificateurs dans la Table.

En fait, Knuth (1) démontre qu'à la condition que les clés des enregistrements soient entrés dans un ordre aléatoire, la longueur moyenne d'un sommet de l'arbre à la racine (en d'autres termes, le nombre de comparaisons nécessaires pour une recherche fructueuse ou infructueuse) est de l'ordre de 2 ln N ($\approx 1.386 \log_2 N$), si N désigne le nombre d'enregistrements dans la Table (ou encore le # de l'ensemble des sommets de l'arbre).

Ainsi, $\boxed{C_N = C'_N = 2 \ln N}$

Dès lors, la longueur de la recherche reste intolérable dès que N > 30. C'est pourquoi Gries (2), tirant parti de ce qu'il n'importe pas, dans cette méthode, que les enregistrements soient contigus en Mémoire Centrale, propose de réduire le Temps de recherche en augmentant le nombre d'arbres. Ainsi partitionne-t-il l'ensemble des identificateurs d'un programme FORTRAN en 6 classes, selon le nombre de leurs caractères (min 1, max 6). A chacune de ces classes, correspond un arbre binaire sur la racine duquel pointe une variable (RACINE 1, ... RACINE 6).

Soit, par exemple, à entrer dans une Table la suite d'identificateurs C, D, MN, RZ, ROJ, A, BA, MNQ, END, B, CE.
Sur le schéma de gauche, Fig. 3.2.3.3. (a), nous supposerons que nous n'avons à notre disposition qu'un seul arbre binaire; sur le schéma de droite, au contraire, nous en aurons un pour chaque classe d'identificateurs.

---

(1) KNUTH, Art of Programming, III, p. 427.

(2) GRIES, Compiler Construction, p. 224.

Au prix d'un calcul opéré par le SCANNER, il ne fait guère de doute que le Temps de recherche moyen est nettement minimisé sur la figure de droite.

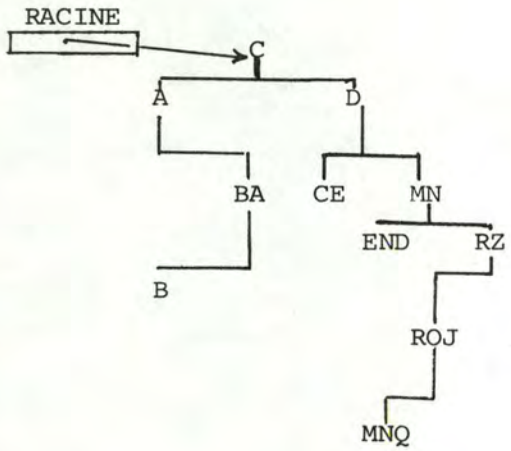


Fig. 3.2.3.3. (a)

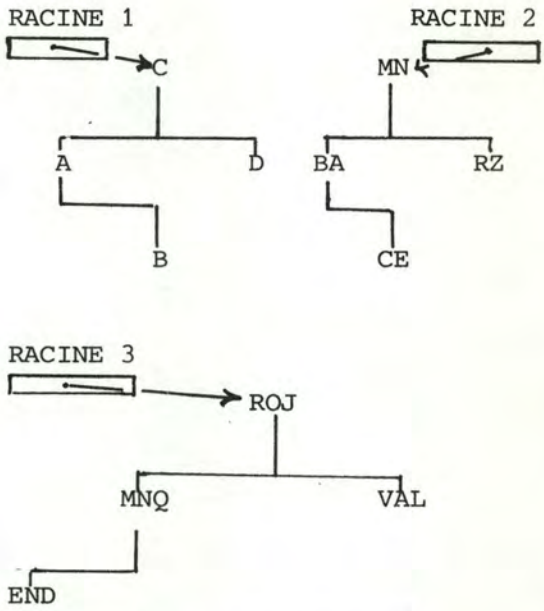


Fig. 3.2.3.3. (b)

b) Place-mémoire

Supposons que nous ne traitions que des enregistrements de longueur fixe M, que nous en ayons introduit un nombre N dans la Table T. Il appert à l'évidence que la place occupée = N x M (Recherche séquentielle - dichotomique).

Si maintenant nous adjoignons à chaque enregistrement 2 positions de longueur p (Recherche par arbre binaire), la taille-mémoire requise pour les N enregistrements de la Table T devient égale à

$$\boxed{N * (M + 2p) \text{ ou encore } (N * M) + (2N * p)}$$

Par rapport à l'allocation minimale $N * M$, la mémoire perdue $= \boxed{\frac{2P}{M}}$

Dès lors, si M est grand en regard de 2 p, la perte de place est minime; si, en revanche, M est petit face à 2 p, alors le rapport $\frac{2P}{M}$ grandit et la perte de place se révèle considérable.

Or, l'enregistrement logique d'une Table de Symboles est petit par rapport à 2 p ($M \sim 2$ mots/mémoire). Si bien que la perte de place encourue diminue, notablement, la performance globale de cette méthode.

3.3. Recherche à partir des propriétés digitales d'une clé (HASH CODING)

3.3.1. Introduction

Ces méthodes de recherche se proposent de calculer une fonction F dont la valeur F(K) pour un argument K donné serait l'adresse relative au début de la Table (numéro d'entrée) de l'enregistrement E_K de clé K.

Idéalement, F devrait être bijective, ou encore

$$\forall\ K_1 \neq K_2 \Rightarrow F(K_1) \neq F(K_2).$$

Mais, eu égard au nombre de clés possibles et supposées équiprobables, cette proposition impliquerait, par exemple, que le nombre d'entrées de la Table, désignons-le par S, égale le cardinal de l'ensemble des clés possibles.

$$S = \#\ \{\text{clés possibles}\}\ !$$

Si bien qu'en pratique, la contrainte de bijection est abandonnée, c'est-à-dire :

$$\boxed{\exists\ K_1 \neq K_2 : F(K_1) = F(K_2)}$$

Cet événement s'appelle COLLISION (KNUTH), OVERFLOW (HOPGOOD), SYNONYME (CHERTON).

Nous aurons donc, dans les paragraphes suivants, deux points principaux à envisager :

1) la fonction F (nommée fonction HASH) : nous décrirons ses caractéristiques, les principales méthodes employées pour la calculer et leurs performances.
2) la manière de résoudre les COLLISIONS, qui partitionne, comme nous le verrons, l'ensemble des méthodes de HASH en deux grandes classes, selon qu'elles résolvent ce problème par chaînage ou non.

3.3.2. La Fonction F

3.3.2.1. Préalables

Le domaine de la fonction F appartient, par définition, à N; en d'autres mots, tout argument de F (clé) doit être de classe numérique.

Or, la plupart du temps, les identificateurs (clés) appartiennent aux classes alphabétique ou alphanumérique . Si le calculateur est binaire, aucune difficulté ne surgit, puisque rien ne l'empêche de considérer comme numérique la représentation interne des symboles. Si, au contraire, le calculateur est décimal, il devient nécessaire, avant toute autre action, de transformer les clés alphabétiques ou alphanumériques en clés numériques. A ce propos, plusieurs schémas d'encodage ont fait l'objet d'essais prolongés. A l'expérience, il paraît que les meilleurs résultats ont été obtenus en transformant les lettres a, b, ... z en nombres décimaux 11.....36 (1).
Cette première transformation opérée, il conviendra, par souci d'efficacité (travail uniquement par des instructions RR), de ramener la longueur de la clé à 1 mot-mémoire, sans en altérer l'originalité. Cette dernière hypothèse implique qu'une simple troncature à droite, par exemple, serait mauvaise. En effet, les identificateurs XYZAB, XYZAR, XYZAG deviendraient ainsi XYZA et en conséquence F(XYZAB) = F(XYZAR) = F(XYZAG). La fonction F assurerait donc une mauvaise distribution des clés par entrée de la Table.

En général, deux techniques sont utilisées. Toutes deux divisent d'abord la clé en sections d'une longueur d'un mot-mémoire. La première additionne ensuite les différentes sections et ignore le report; la seconde applique aux différents mots ainsi créés l'instruction OU EXCLUSIF.
Ainsi, supposons que, dans un calculateur binaire, nous rencontrions l'identificateur CICERON.

PAS 1 : C I C E R O N b

En représentation interne, selon le code ECBDIC, nous trouverons :

C I C E : 1 1 0 0 0 0 1 1 1 1 0 0 1 0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1
R O N b : 1 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 1 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0

(1) Méthode proposée par LUM, Transform Techniques, p. 231.

En utilisant la technique d'addition nous obtiendrons :

| PAS 2 | : Résultat : 1001 1101 1010 0000 1001 1001 0000 0101

tandis que l'application de l'instruction OUX donnera pour résultat :

| PAS 2 | : Résultat : 0001 1010 0001 1111 0001 0110 1000 0101.

3.3.2.2. Description des différentes méthodes

Nous allons successivement étudier trois des méthodes de transformation parmi les plus classiques : la division, la multiplication et le folding (1).

3.3.2.2.1. La Division (Compilateur IBM, PL/1 (F))

Dans cette méthode, la clé K est divisée par M, la taille logique de la Table (nombre d'entrées) et le reste de la division donne l'entier i, adresse logique (numéro d'entrée) de l'enregistrement associé à cette clé (2). Plus formellement F(K) = i = K mod M. Naturellement, le choix de M tiendra compte de ce que la distribution des clés par adresses doit être aussi uniforme que possible. A cet égard, le sens commun - confirmé par l'expérience ! - rejettera, par exemple,un nombre M pair car le reste de la division par un nombre pair a la parité du dividende.
Si bien qu'en général, M sera choisi parmi les nombres premiers (M=211 dans le compilateur PL/1 (F)).
A titre d'exemple, supposons que nous voulions construire une Table de 97 entrées, que les clés introduites soient numériques et compor-

---

(1) Cfr. pour bibliographie LUM, Transform Techniques, pp. 229-230; KNUTH, Art of Programming, III, pp. 506 - 513, GRIES, Compiler Construction, p. 223; HOPGOOD, Compiling Techniques, p. 27, etc...

(2) La méthode, appelée codage algébrique (algebraic coding), procède de la même idée mais le diviseur, au lieu d'un entier, est un polynôme modulo 2. Au dire même de Knuth, (KNUTH, Art of Programming, III, p. 503), cette technique convient mieux aux problèmes de matériel et de microprogrammation qu'aux problèmes de logiciel. Pour une information plus approfondie et plus complète, il conviendra de consulter KNUTH, Art of Programming III, pp. 513 et 688; LUM, Transform Techniques, p. 230 ainsi que M. HANAN et autres, IBM Journal Research & Development 7 (1963), pp. 121 - 129.

tent de 1 à 4 digits décimaux, alors l'adresse logique de l'enregistrement associé à la clé 6525 est

6 5 2 5 mod 97 = (26)

3.3.2.2.2. La Multiplication (MIDDLE SQUARE METHOD) (1)

La clé K est multipliée par elle-même (élevée au carré) et l'adresse logique résulte de la troncature des digits extrêmes jusqu'à ce que le nombre de digits restants soit égal à la longueur de l'adresse logique souhaitée.

Reprenons l'exemple précédent.

PAS 1 : Elevons la clé au carré 6 5 2 5 * 6 5 2 5 = 3415625

PAS 2 : Tronquons logiquement la clé en alternant digit extrême gauche - digit extrême droit, jusqu'à ce qu'il n'en reste que 2. L'entrée recherchée aura donc le numéro (56).

3.3.2.2.3. Le Folding

Dans cette technique, la clé K est d'abord divisée en un certain nombre de parties, chacune d'elles, excepté peut-être la dernière, ayant une longueur égale à la longueur de l'adresse logique recherchée.

A partir de ce moment, deux méthodes de transformation peuvent être mises en oeuvre.

La première consiste à "replier" (FOLD-BOUNDARY) les différentes parties de la clé les unes au-dessus des autres et à additionner les digits occupant des positions identiques; le report éventuel de l'addition est ignoré.

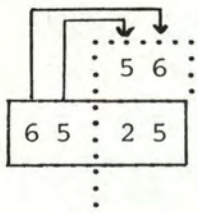
Montrons, sur notre exemple, le fonctionnement de cette méthode.

PAS 1 : Partition de la clé 6525 en sections de 2 digits décimaux

65 : 25

PAS 2 : "Folding boundary"

(1) Knuth, s'appuyant sur les travaux de R.W. Floyd, soutient qu'il est également possible d'obtenir une bonne fonction Hash en multipliant la clé K par une constante A, convenablement choisie. Il en discute longuement les caractéristiques . cfr. KNUTH, Art of Programming, III, pp. 508 - 513.

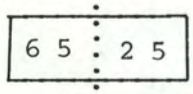


PAS 3 : Addition

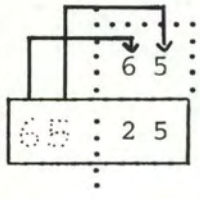
c'est-à-dire le numéro d'entrée recherché.

La seconde fait glisser les sections initiales (FOLD SHIFT) de manière à ce que leurs bornes soient alignées avant l'addition; dans ce cas aussi, le report éventuel est ignoré. Nous aurons donc le déroulement suivant, si l'on considère la clé 6525.

PAS 1 :

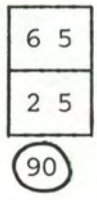


PAS 2 :



shift et alignement des bornes inférieures.

PAS 3 : Addition



, numéro d'entrée de la Table.

3.3.2.3. Analyse des performances de ces méthodes

Une bonne fonction Hash doit satisfaire à deux exigences (1) :

1) Son calcul sera très rapide; cette propriété cependant dépend quelque peu de la machine et ne prend de l'importance qu'au moment où le temps de calcul est significatif par rapport au temps d'accès, c'est-à-dire, pour l'essentiel, lorsque les données à accéder sont rangées en Mémoire Centrale.

---

(1) KNUTH, Art of Programming, III, p. 512.

2) Elle minimise les collisions; autrement dit, elle répartit bien les clés selon les entrées de la Table.

C'est à cet aspect du problème que se sont intéressés Lum, Yuen et Dodd (1).

Leur étude a porté sur des fichiers réels dont les clés étaient de différentes classes (numérique, alphanumérique, alphabétique) et de longueurs diverses (5, 6, 10 ou 12 symboles). Les auteurs ont cherché à connaître le nombre d'accès nécessaires pour rechercher un enregistrement dont la clé était spécifiée (Recherche fructueuse).

En toute généralité, cette performance dépend de trois variables :

1) le facteur de chargement de la Table ($\alpha : 0 \leq \alpha \leq 1$), c'est-à-dire le nombre d'enregistrements effectivement rangés dans une Zone-Mémoire divisé par le nombre d'enregistrements que l'on peut y ranger. W.W. Peterson (2) a démontré, en effet, que le nombre d'accès nécessaires pour retrouver un enregistrement dans une Table était indépendant de la taille de la Table et de l'ordre d'entrée des enregistrements mais croissait avec le facteur de chargement de cette Table.

2) Les méthodes de résolution des collisions désignées par CHAINAGE ou SANS CHAINAGE. Nous verrons plus loin, (3) la signification réelle de ces deux techniques.

3) La fonction hash elle-même (Division, multiplication, Folding).

A première vue, il paraît malaisé d'isoler un de ces éléments. En fait, il n'en est rien. En effet, si, quel que soit le facteur de chargement, quelle que soit la méthode de résolution des collisions, l'écart entre deux lignes du Tableau Fig. 3.3.2.3. reste constant ou va grandissant, alors, il est naturel de conclure que cet écart est dû à la seule fonction Hash, puisqu'il s'agit du seul élément variant d'une ligne à l'autre. Dès lors même, nous avons trouvé une mesure des performances de ces différentes méthodes de transformation.

---

(1) LUM, Transform Techniques.

(2) W.W. PETERSON, Random-access ,pp. 130 - 146, passim.

(3) Cfr. p. 84 et p. 97.

Facteur Chargem. α	0.50		0,60		0.70		0.80		0.90	
Méthode de Transform. \ Collisions	Chaînage	Open	Chaînage	Open	Chaînage	Open	Chaînage	Open	Chaînage	Open
DIVISION	1.19	4.52	1.25	5.04	1.28	4.73	1.34	10.10	1.38	22.42
MULTIPLICATION (Midsquare)	1.26	1.73	1.32	3.17	1.36	3.85	1.41	13.25	1.45	27.14
FOLDING BOUNDARY	1.39	22.97	1.44	25.36	1.45	23.95	1.51	41.10	1.55	69.63
FOLDING SHIFT	1.33	21.75	1.52	28.57	1.40	39.00	1.48	66.26	1.40	77.01

Fig. 3.3.2.3. Tableau comparatif des méthodes de transformation de clés

Conclusions de cette étude

1) La division donne presque toujours les meilleurs résultats.
2) La multiplication (midsquare) a de bonnes performances aussi longtemps que le facteur de chargement n'excède pas 0.70.
   A son actif, nous mentionnerons encore que l'écart-type de sa distribution est le plus petit de tous (0.03), ce qui signifie que les résultats sont bien groupés autour de la moyenne.
   Néanmoins, il faut inscrire à son passif, des résultats franchement médiocres, si les clés commencent ou se terminent par une suite de 0 (1).
3) Enfin - et le rappel n'est pas de pure forme - ces résultats n'ont de valeur que dans le contexte de notre problème (enregistrements rangés en Mémoire Centrale). Pour s'en convaincre, le lecteur voudra bien se reporter à l'article de Lum (2).

---

(1) KNUTH, Art of Programming, III, p. 508.

(2) LUM, Transform Techniques.

3.3.3. Résolution des collisions

Supposons que pour deux clés K_1 et K_2, telles que $K_1 \neq K_2$, $F(K_1) = F(K_2)$; il se produit donc une collision. Deux grandes méthodes ont été, au fil des ans, mises au point pour résoudre ce problème. La première (HASH avec chaînage) explicite par des pointeurs l'itinéraire à suivre pour retrouver ou insérer un enregistrement, tandis que la seconde (HASH sans chaînage) édicte des règles qui construisent implicitement cet itinéraire.
Cette différence est caractéristique.

3.3.3.1. Hash avec chaînage

3.3.3.1.1. Description générale

La manière sans doute la plus simple de résoudre le problème des collisions consiste à créer autant de listes qu'il y a d'entrées dans la Table des Symboles. Cette hypothèse implique, évidemment, qu'il faut inclure, dans chaque enregistrement, un pointeur.
Dès lors, la procédure se déroule, logiquement, de la façon suivante pour un enregistrement E_K de clé K:

1) Calcul de F(K);
2) Recherche séquentielle de l'enregistrement E_K à travers la liste de numéro F(K) + 1, si les entrées de la Table sont numérotées de 1 à M.

Pratiquement, deux techniques sont utilisées : le chaînage séparé et le chaînage dans la Table.

3.3.3.1.2. Chaînage séparé

1. Description générale (1)

Imaginons que nous disposions d'une Table de M pointeurs, initialisés à 0, d'une zone de mémoire initialement vide et d'un pointeur appelé LIBRE, par exemple.
Les enregistrements sont présumés de longueur fixe (sans que cette condition soit nécessaire) et chacun d'eux comporte les 3 champs suivants : CLE, VALEUR (cette position contient les renseignements tra-

---

(1) GRIES, Compiler Construction, pp. 220 - 222, passim.

ditionnellement rangés dans une Table de Symboles et dont nous avons discuté précédemment), POINT (qui indiquera l'enregistrement suivant dans la liste).

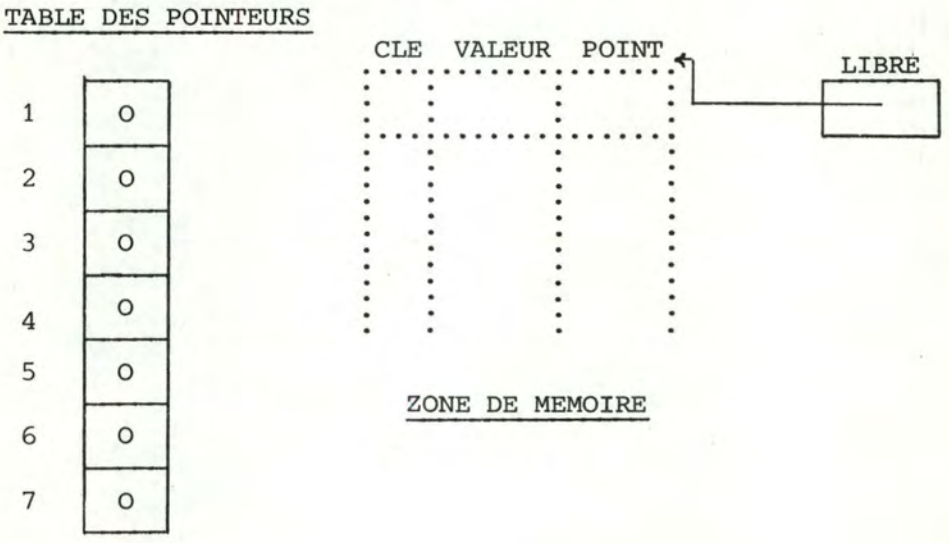


Fig. 3.3.3.1.2. (a) Situation initiale

Supposons que nous voulions entrer un enregistrement E_K de clé K dans la Table.

Nous procéderons comme suit :

1) Calcul de F(K);
2) L'entrée de la Table des pointeurs obtenue (par exemple la quatrième) = O
3) Dès lors, nous ajoutons 1 x la longueur de l'enregistrement physique à LIBRE
4) Nous insérons l'enregistrement E_K à l'adresse délivrée par LIBRE
5) Nous plaçons le contenu de LIBRE dans la quatrième entrée de la Table des pointeurs.

Voici comment, dès lors, se présente la situation :

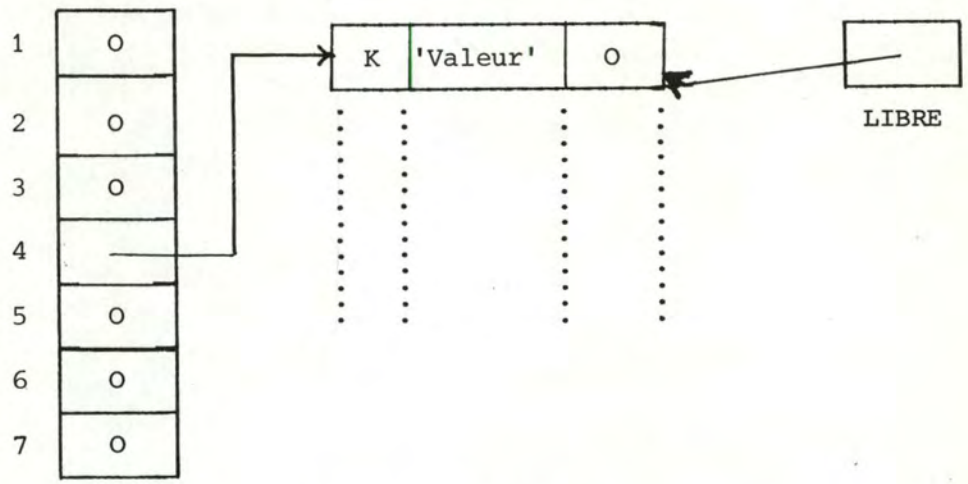


Fig. 3.3.1.2. (b)

Supposons, à présent, qu'après 3 essais nous ayons le schéma suivant (Fig. 3.3.3.1.2 (c)) et que nous voulions entrer dans la Table des Symboles, l'enregistrement E_M de clé M; par malchance, F(M) = 3; une collision surgit donc.

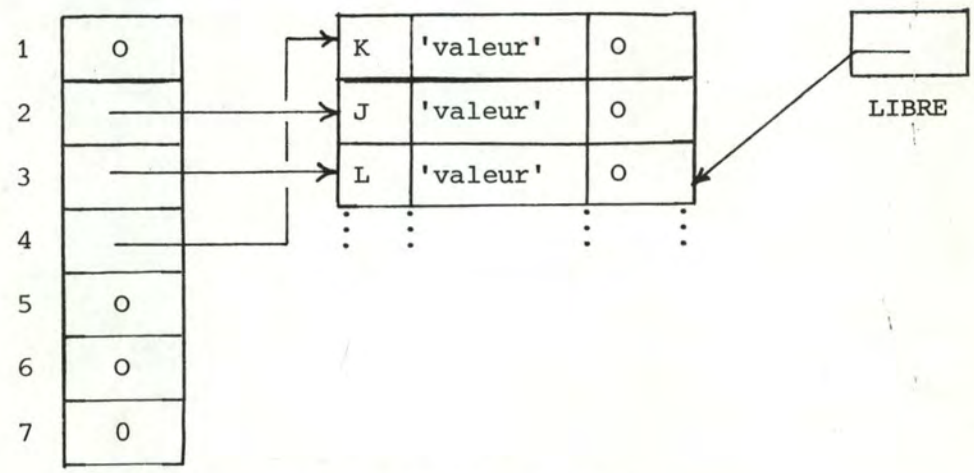


Fig. 3.3.3.1.2. (c)

Dans cette méthode, E_M sera implanté à la suite de l'enregistrement E_1 et la liste n° 3 sera donc mise à jour. Si bien que nous obtiendrons la configuration suivante :

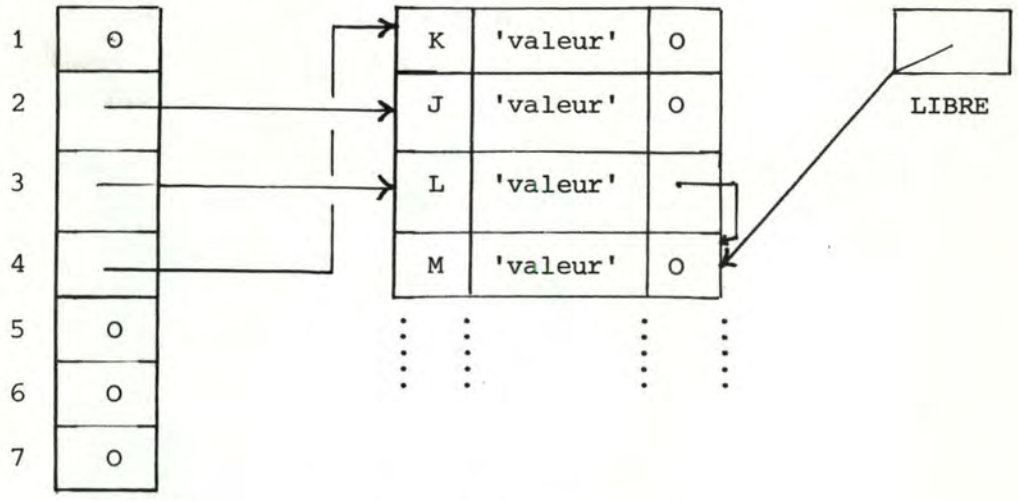


Fig. 3.3.3.1.2. (d)

Nous allons voir que l'algorithme suivant dû à Gries (1), structure de cette façon l'accès à la Table des Symboles.

2. Algorithme d'implémentation

Etant donné une Table de M pointeurs (TABLE POINT) dont les entrées sont numérotées de 1 à M, une zone de Mémoire et un pointeur LIBRE contenant l'adresse de cette zone-1, la procédure se déroule comme suit pour un enregistrement E_K de clé K.

PAS	ALGORITHME (1)	PAS	COMMENTAIRES
1	P1: = Adresse (TABLE POINT) + F(K) + 1 P2: = Contenu (P1)	1	P1 et P2 sont des pointeurs initialisés à chaque entrée dans la routine, le premier contient l'adresse de la Table des Pointeurs à tester, le second (P2) le contenu de la case-mémoire correspondant à cette adresse (O ou une adresse.

(1) GRIES, Compiler Construction, p. 222. Morris a écrit en FORTRAN un programme assez comparable. Cfr. MORRIS, Scatter Storage , pp. 43-44.

2	Si P2 = 0, aller en 4 Sinon, comparer la clé K de l'enregistrement à insérer à la clé, ~~disons S~~, de l'enregistrement d'adresse P2. (P2.CLÉ)	... et entreprendre la procédure d'insertion.
3	Si S^{K} = $K^{P2.CLE}$, la recherche est fructueuse; Sinon P1 := ~~P2~~ Adresse (P2. POINT) P2 := Contenu (P1). POINT Aller en 2.	La recherche se poursuit à travers la liste de numéro F(K) +1. A cet effet, P1 est garni de l'adresse du champ POINT de l'enregistrement E_S ~~de la clé S~~ d'adresse P2 et P2 contient la valeur de ce champ (0 ou une adresse).
4	LIBRE: = LIBRE + 1; P2 := LIBRE; Rangement de l'enregistrement E_K (K, 'valeur', O) à l'adresse Contenu (P2). Contenu (P1). ~~POINT~~ := LIBRE	Mise à jour du pointeur LIBRE. Mise à jour des listes.

3. Performances

A. Temps logique

En moyenne, le nombre d'accès requis dans l'hypothèse d'une recherche fructueuse sera

$$C_N \approx 1 + \frac{\alpha}{2}$$

(1) si α représente le facteur de chargement

(1) KNUTH, Art of Programming, III, p. 535.

et $C'_N \approx 2$ → Knuth donne uniquement le tableau des résultats!

Ces résultats sont illustrés par le tableau Fig. 3.3.3.1.2.(e) et repris par les graphiques Fig. 3.3.3.1.2. (f et g).

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Recherche fructueuse	1.05	1.10	1.15	1.20	1.25	1.30	1.35	1.40	1.45
Recherche infructueuse	1.0048	1.0187	1.0408	1.0703	1.1065	1.1488	1.197	1.249	1.307

Fig. 3.3.3.1.2. (e)

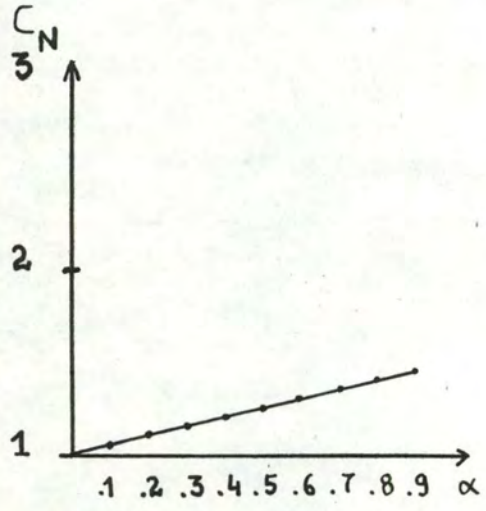


Fig. 3.3.3.1.2.(f) Recherche fructueuse

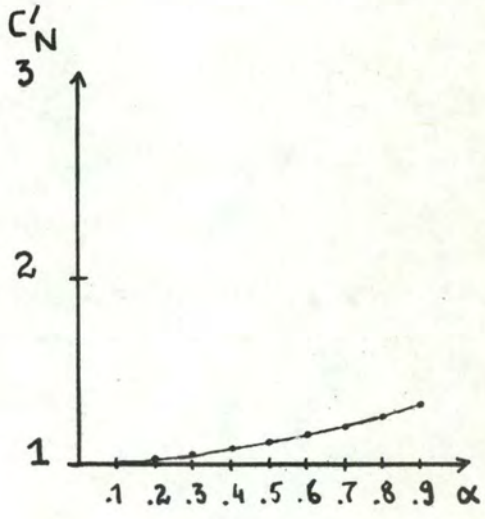


Fig.3.3.3.1.2.(g) Recherche infructueuse

B. Place-Mémoire

Si, selon la technique de Hash avec chaînage, nous voulons ranger en Mémoire Centrale M enregistrements logiques de longueur Z, sachant que chaque pointeur est d'une longueur p, la place totale nécessaire N sera :

N := M * Z + M * p + Q * p où Q représente le nombre d'entrées de la Table des pointeurs.

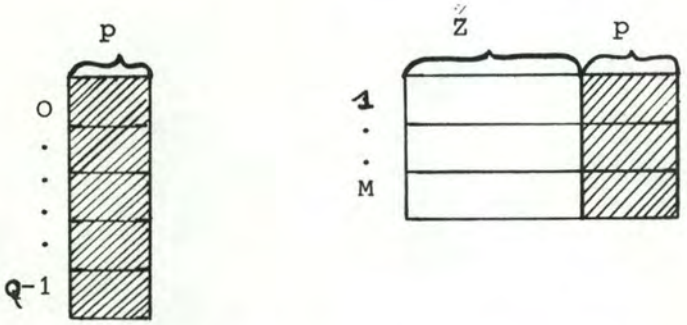


Table des Pointeurs Zone des Enregistrements

Fig. 3.3.3.1.2. (h)

Le nombre Q est quelconque mais fixé avant la compilation, puisque l'ensemble des numéros d'entrée de la Table des Pointeurs égale l'ensemble des valeurs délivrées par la fonction F. Donc Q * P est, a priori, retenu et la tentation est grande de choisir Q le plus petit possible. Il est immédiat néanmoins que plus ce nombre est petit par rapport au nombre d'enregistrements à entrer, plus la recherche d'un enregistrement s'allonge ! Nous reviendrons plus en détail sur le choix de ce nombre d'entrées(1) où le problème présente, du reste, un caractère plus aigu .

Au total, par rapport à la place minimale requise M * Z, la perte de place encourue (hachurée sur notre croquis) égale

$$M * P + Q * P. = (M + Q) * P$$

(1) Cfr. p. 102.

3.3.3.1.3. Chaînage dans la table (1)

1. Description générale

Cette méthode se sert d'une Table (TABLE) de M entrées, numérotées de 1 à M. Chacune d'elles est initialisée de manière à pouvoir déterminer si cette entrée est déjà occupée. Enfin, chaque enregistrement est supposé de longueur fixe et se présente ainsi :

LIBRE	CLE	VALEUR	POINT

La position LIBRE indique par '+' que l'entrée est occupée, par '-' qu'elle est libre.

Supposons que M = 7 et que nous voulions insérer un enregistrement E_K de clé K telle que F(K) + 1 = 4;(2) la Table se présente donc de la sorte :

1	-			
2	-			
3	-			
4	+	K	'valeur'	O
5	-			
6	-			
7	-			

Fig. 3.3.3.1.3. (a)

Imaginons, comme dans l'exemple précédent, qu'après 3 essais nous ayons le schéma décrit Fig. 3.3.3.1.3.(b) et que nous désirions entrer dans la Table l'enregistrement E_m de clé M, telle que F(M) + 1 = F(L) + 1= 3.

---

(1) KNUTH, Art of Programming, pp. 514 - 515 passim. Sa source est F.A. WILLIAMS CACM.2, Juin 1959, pp. 21 - 24.

(2) Rappelons encore que F(K) prend des valeurs $\in \{0, \ldots M-1\}$ alors que nous avons pris pour convention de numéroter les entrées, dans ce cas-ci, de 1 à M.

	LIBRE	CLE	VALEUR	POINT
1	-			
2	+	J	'valeur'	0
3	+	L	'valeur'	0
4	+	K	'valeur'	0
5	-			
6	-			
7	-			

Fig. 3.3.3.1.3.(b)

	LIBRE	CLE	VALEUR	POINT
1	-			
2	+	J	'valeur'	0
3	+	L	'valeur'	•
4	+	K	'valeur'	0
5	-			
6	-			
7	+	M	'valeur'	0

Fig. 3.3.3.1.3.(c)

Pour assigner une entrée à l'enregistrement E_m, initialisons au premier appel de la routine, une variable de travail, soit R, à la valeur M + 1 (en l'occurrence 8) et décrémentons-la de 1 jusqu'à ce qu'une entrée vide soit trouvée. Il est donc toujours vrai que TABLE [j] est occupée $\forall$ j : R $\leq$ j $\leq$ M.

La procédure d'insertion se déclenche alors et le pointeur POINT correspondant à la 3e entrée est mis à jour. Le résultat apparaît Fig. 3.3.3.1.3. (c).

Lorsque la Table est pleine, il suffit de demander à l'Allocateur de Mémoire une zone supplémentaire où seront rangés les nouveaux enregistrements. En général, on appellera cette zone, zone d'OVERFLOW.

2. Algorithme d'implémentation

Etant donné une Table (TABLE) de M entrées, numérotées de 1 à M, marquées '+' ou '-' ,selon qu'elles sont occupées ou vides, et une variable de travail R initialisée à M + 1, l'algorithme suivant structure l'accès de la Table de la manière que nous venons de décrire.

PAS	ALGORITHME	COMMENTAIRES
1	I := F(K) + 1	La valeur de F(K) + 1 est rangée dans la variable de travail I.
2	Si TABLE [I] est vide, aller en 6.	Sinon, TABLE [I] est occupée et nous allons suivre la liste des entrées occupées à partir de l'entrée TABLE [I].
3	Si K = CLE [I] , la recherche se termine avec succès, sinon aller en 4.	
4	Si POINT [I] ≠ 0, I = POINT[I]; I, aller en 3.	Si POINT [I] = 0, le parcours de la liste est terminé, sans avoir rencontré l'enregistrement E_K de clé K. Il faut dès lors, insérer l'enregistrement E_K à la "première" place "libre".(PAS 5)
5	R := R - 1; Si R = 0, appeler la procédure 'Table pleine'; Si LIBRE [R] est 'occupé', aller en 5 ; Sinon POINT [I] :=R ; I := R;	Décrémenter la variable de travail R jusqu'à ce que soit trouvée une entrée libre. 'Table pleine" = Mise à disposition d'une zone d'Overflow. Préparation de la procédure de chaînage de l'enregistrement E_K à la liste des enregistrements de clé S telle que F(S) + 1 = F(K) + 1.
6	LIBRE [I] := '+'; CLE [I] := K; POINT [I] := 0.	Insertion de l'enregistrement E_K de clé K.

3. Performances

a) Temps logique

En moyenne, le nombre d'enregistrements accédés dans le cadre d'une recherche fructueuse sera

$$C_N \approx 1 + \frac{1}{8\alpha} (e^{2\alpha} - 1 - 2\alpha) + \frac{1}{4}\alpha$$

tandis que, dans l'hypothèse d'une recherche infructueuse, ce nombre moyen deviendra :

$$C'_N \approx 1 + \frac{1}{4} (e^{2\alpha} - 1 - 2\alpha)$$

Ces expressions sont calculées, pour certaines valeurs de α, dans le tableau Fig. 3.3.3.1.3. (d). Les résultats sont repris sur les graphiques Fig. 3.3.3.1.3.(e).

α	0.1	0.2	0.3	0.4	0,5	0,6	0.7	0.8	0.9
C_N	1.061	1.128	1.196	1.266	1.304	1.380	1.470	1.550	1.676
C'_N	1.005	1.022	1.055	1.106	1.179	1.270	1.413	1.560	1.812

Fig. 3.3.3.1.3. (d) Tableau des résultats

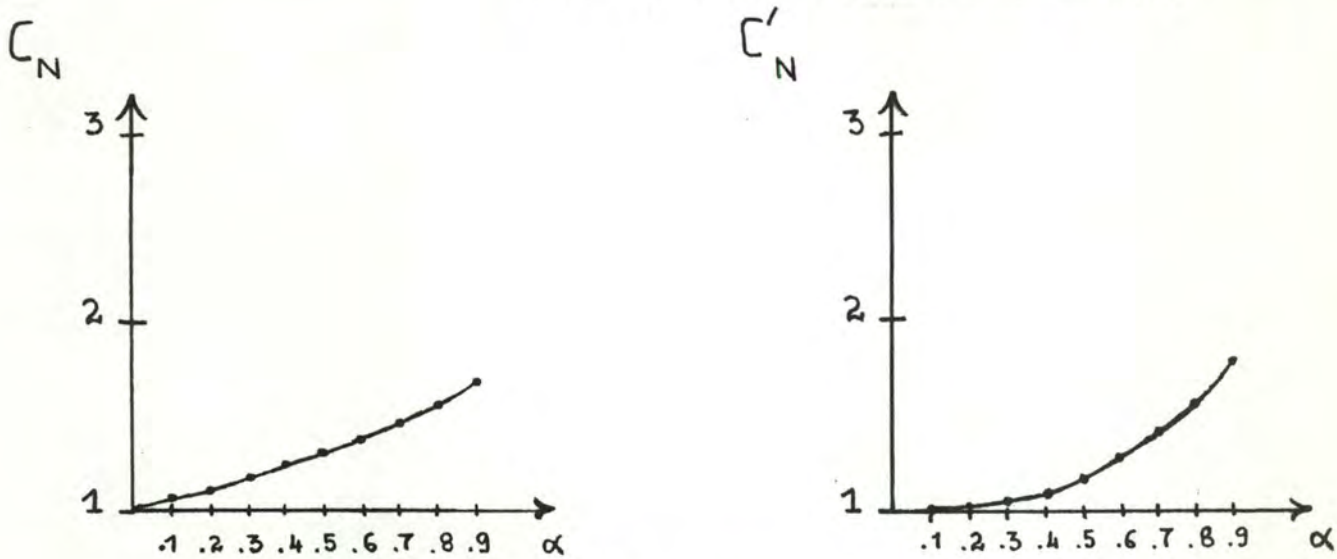


Fig. 3.3.3.1.3. (e) Graphiques correspondant au tableau des résultats.

b) Place-Mémoire

Désignons par Q le nombre d'entrées de la Table. Comme nous l'avons dit précédemment (1), ce nombre est connu a priori; soit Z la longueur d'un enregistrement logique et p la longueur d'un pointeur.

Au total, la place a priori retenue égale donc, dans ce cas, Q * (Z + P).

Supposons que nous désirions ranger dans la Table M enregistrements.

Deux cas sont à envisager :

1) M < Q

Dans cette hypothèse, la place totale retenue = Q * (Z + P)

la place minimale requise = M * Z

la perte de place encourue = Q * P + (Q - M) * Z

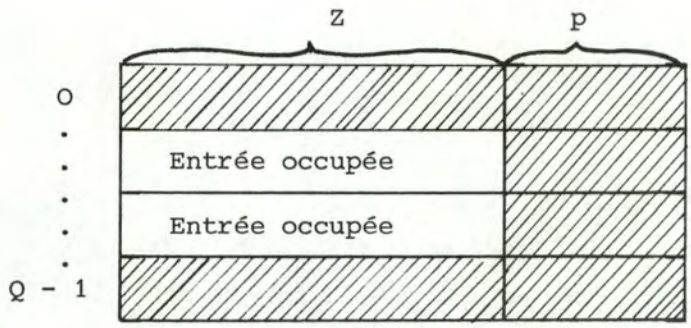


Fig. 3.3.3.1.3.(f) Perte de place encourue si M = 2.

2) M ≥ Q : lorsque la Table est pleine, rangement séquentiel des enregistrements suivants dans une zone d'Overflow.

Alors, la place totale occupée = M * (Z + P)

la place minimale requise = M * Z

la perte de place retenue = M * P, c'est-à-dire la place pour M pointeurs.

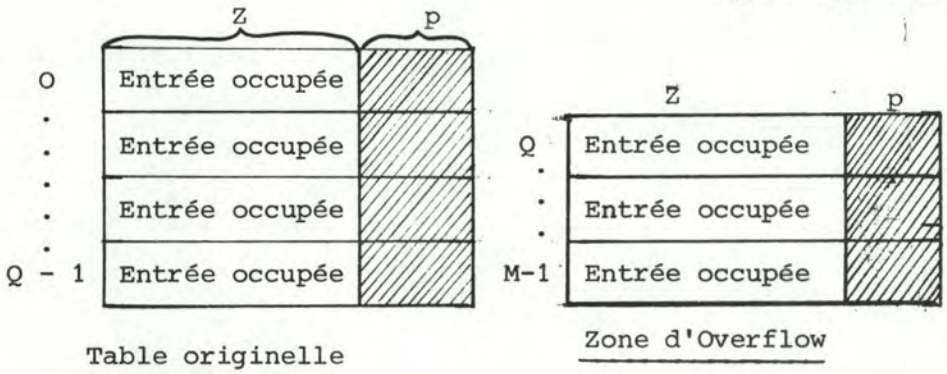


Fig. 3.3.3.1.3. (g)

(1) Cfr. p. 90

Au total, par rapport à l'allocation de place minimale, la perte de place qui résulte de l'utilisation de cette méthode s'avère au moins égale à Q * P, ∀ M le nombre d'enregistrements logiques.

3.3.3.1.4. Conclusions sur les méthodes Hash avec chaînage

Si l'on compare les performances des deux grandes méthodes que nous avons analysées, on s'aperçoit :

1) que du point de vue temps logique, la méthode de chaînage hors de la Table l'emporte, comme le montrent les graphiques récapitulatifs Fig. 3.3.3.1.4.

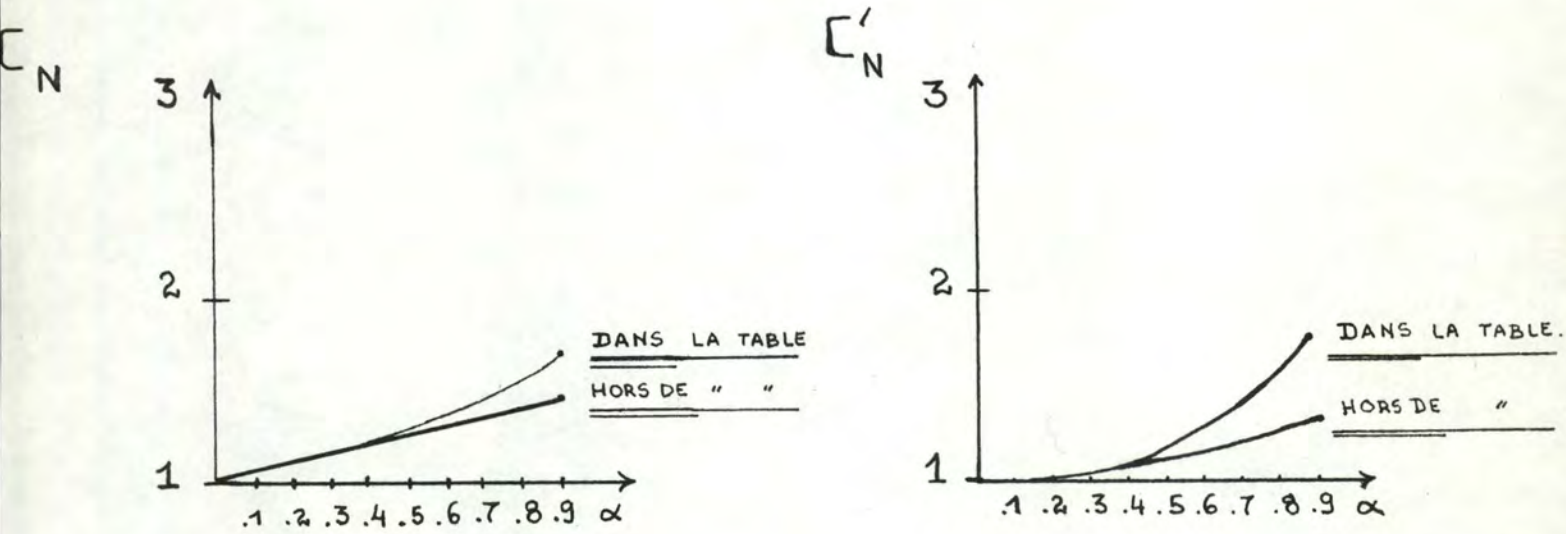


Fig. 3.3.3.1.4. Graphiques récapitulatifs (1)

2) que du point de vue place-mémoire, la méthode de chaînage dans la Table est la meilleure, sans qu'il soit possible de mesurer avec précision à quel niveau se situe le gain de place ainsi réalisé.

---

(1) On trouvera le détail pp. 89 et 94.

3.3.3.2. Hash sans chaînage

Depuis quelques années, les efforts des chercheurs se sont orientés dans cette voie et de nombreuses variantes ont fait florès dans la littérature. Dès lors, nous paraît-il indispensable de décrire d'abord le principe général de cette méthode ainsi que les performances qui peuvent en être attendues et d'envisager ensuite les principales solutions proposées. (1)

3.3.3.2.1. Principe général

1. Description générale

A/ Sens général de la démarche

Soit, par exemple, une Table T (Fig. 3.3.3.2.1.(a)) de 8 entrées de même longueur - la condition est nécessaire - notées TABLE [0].... TABLE [7] .

L'ensemble de ces entrées est partitionné en deux classes discernables : la classe des entrées vides (premier item (-) sur notre exemple) et la classe des entrées occupées (premier item (+)).

	LIBRE	CLE	VALEUR
0	-		
1	-		
2	-		
3	-		
4	-		
5	-		
6	-		
7	-		

Fig. 3.3.3.2.1.(a) Situation initiale de Table T.

	LIBRE	CLE	VALEUR
0	-		
1	+	J	'valeur'
2	-		
3	+	L	'valeur'
4	-		
5	+	M	'valeur'
6	-		
7	+	S	'valeur'

K	'valeur'

E_K

Fig. 3.3.3.2.1.(b) Situation de Table T avant l'insertion de l'enregistrement E_K de clé K.

(1) Les principales, et non toutes les solutions proposées.

Supposons que dans l'état où elle se trouve Fig. 3.3.3.2.1.(b), nous voulions y ranger un enregistrement E_K de clé K. Il importe d'abord de calculer, comme d'habitude, F(K), de tester ensuite la classe à laquelle appartient la Table [F(K)] .

Si cette entrée est vide [F(K)] $\in \{0,2,4,6\}$, l'enregistrement E_K est inséré; sinon [F(K) $\in \{1,3,5,7\}$], on se propose de parcourir, sans indication physique du chemin (pointeurs), les différentes entrées de la Table jusqu'à ce qu'on ait trouvé une entrée occupée de clé K (Recherche fructueuse);

- on ait trouvé une entrée vide (Recherche infructueuse); En pareille hypothèse, il convient, évidemment, que nous puissions conclure que l'enregistrement E_K de clé K n'est pas déjà rangé dans la Table.
- on soit revenu à l'entrée de départ; la Table est alors considérée comme pleine.

Tout le problème revient donc à formuler une règle qui, de manière déterministe, à partir d'une clé K donnée, produira une séquence d'essais, c'est-à-dire une suite d'entrées de la Table qui seront inspectées lorsque se déroulera la recherche d'un enregistrement E_K de clé K.

A cet effet, nous ferons appel à une fonction que nous désignerons par F_2.

B/ Définition et caractéristiques de F_2

Généralement, F_2 est une fonction définie :

$$F_2 : i \longrightarrow F_2(i)$$

Son domaine est $\{i : 1 \leq i \leq p - 1\}$ où

- p représente le nombre total d'entrées de la Table et
- i le nombre d'entrées déjà examinées avant la nouvelle (initialement la première) application de F_2.

Dès lors, les choses vont évoluer de la sorte :

- la première entrée de la Table examinée sera celle dont le numéro = $F_1(K)$, pour une clé K donnée;
- la deuxième sera celle dont le numéro = $(F_1(K) + F_2(1)) \bmod p$;
- la nème, celle dont le numéro = $(F_1(K) + F_2(n - 1)) \bmod p$.

Naturellement, les caractéristiques de cette fonction F_2 ne doivent pas détruire l'amélioration des performances dues à l'emploi de la fonction hash initiale (F_1). C'est pourquoi :

1) son calcul sera rapide;
2) la séquence d'essais qu'elle contribuera à produire sera la plus aléatoire possible.
3) cette séquence parcourra la Table sans omission ni redondance.

Nous verrons, tout au long des pages qui suivent, la raison d'être de ces trois caractéristiques essentielles.

Maintenant que nous avons rassemblé tous nos matériaux, il nous est possible de décrire l'algorithme général d'implémentation de hash sans chaînage.

2. Algorithme d'implémentation

Etant donné une Table de M entrées d'égale longueur, notées TABLE [0] ... TABLE [M - 1] - chacune des entrées TABLE [i] contenant un champ [LIBRE] indiquant si elle est vide ou non, un champ [CLE] où est rangée la clé K de l'enregistrement E_K et un champ [VALEUR] où est rangée la valeur associée à la clé K -, l'algorithme suivant construit la Table des Symboles selon la description générale que nous venons d'en faire. La variable N est initialisée à 0 au début d'une compilation.

PAS	ALGORITHME	COMMENTAIRES
1	s := F_1 (K);	Calcul pour la clé K de l'enregistrement E_K de la valeur de la fonction F, selon une des techniques expliquées pp. 79 ssq.; rangement de cette valeur dans la variable entière s. A noter que dorénavant, nous écrirons F_1(K) pour distinguer cette fonction Hash initiale de F_2.
	i := 0;	Initialisation de i (le nombre d'entrées déjà examinées)
	q := s;	La variable de travail q est utilisée pour standardiser l'algorithme à partir du pas 2.

PAS	ALGORITHME	COMMENTAIRES
2	Si TABLE [q] est 'vide', aller en 4; Sinon, si CLE [q] = K, fin;	Double comparaison entraînant les actions suivantes : 1) si l'entrée TABLE [q] est 'vide', nous allons déclencher la procédure d'insertion (pas 4); 2) si l'entrée TABLE [q] est 'occupée' et que CLE [q] = K, la recherche est fructueuse; Sinon, nous passons en 3.

3	i:= i + 1;
	q := s + F_2(i);
	Si q $\geq$ M, q := q - M;
	aller en 2.
		Préparation de l'essai suivant.
		
		Si la valeur de q devient supérieure à M, il faut lui retrancher M (taille logique de la Table), afin de garantir un parcours circulaire de la Table.

4	Si N = M - 1, appel de la procédure 'Table Pleine'; (1) Sinon, N := N + 1; TABLE [q] est 'occupée' et CLE [q] = K.	La variable N sera incrémentée de 1 à chaque insertion et le déclenchement de la procédure 'Table pleine' est prévu quand N = M - 1; de la sorte, il ne sera pas obligatoire d'effectuer, lors de toute recherche, un calcul du nombre d'essais déjà effectués (au plus égal à M - 1), étant donné qu'il reste au moins une position vide. Procédure d'insertion.

(1) L'astuce est de KNUTH, Art of Programming, III, p. 689 - 690.

3. Performances générales

A. Temps logique

Avant de calculer, pour chacune des variantes que nous allons analyser, le nombre moyen d'accès dans le cadre d'une recherche fructueuse ou infructueuse, il nous paraît important d'énoncer trois remarques :

1/ Les performances sont fonction du facteur de chargement de la Table ($0 \leq \alpha \leq 1$), c'est-à-dire du rapport du nombre d'entrées occupées au nombre total d'entrées. (1)

2/ Le calcul de ces performances procède d'un modèle dont l'expérience - nous le verrons - a vigoureusement contesté la validité . Son auteur, Peterson (2) suppose, en effet, que la fonction hash initiale (F_1) répartit uniformément les clés (3) et que l'occupation de chaque entrée est indépendante des autres entrées.

Néanmoins, il nous a semblé bon de faire état des résultats obtenus en raison de la valeur comparative qu'ils recèlent. Les renseignements recueillis offrent, en effet, toutes choses étant égales d'ailleurs, une bonne approximation de certaines qualités des différentes fonctions F_2 ,et il est dès lors permis de les comparer entre elles,sous l'angle du temps logique.

3/ Le lien Temps logique - Temps réel de recherche n'est plus immédiat.

D'autres facteurs entrent en jeu : ainsi, par exemple, les différentes fonctions F_2 réclament un temps de calcul plus ou moins long, qu'il convient d'"ajouter" au Temps logique moyen si l'on désire obtenir une bonne estimation du Temps réel moyen exigé pour une recherche d'un enregistrement quelconque de clé donnée.

---

(1) Cfr. p. 82.

(2) PETERSON, Random-access, I, 1957.

(3) Pour se convaincre du contraire, cfr. le tableau statistique p. 83.

B. Place-Mémoire

I. Place réservée a priori

Toute méthode de résolution des collisions sans chaînage suppose qu'a priori la place soit réservée pour une Table comportant un nombre déterminé d'entrées.

Comment, dans ces conditions, choisir le nombre d'entrées optimal ? Imaginons qu'un concepteur dispose d'un tableau statistique indiquant la dispersion probable du nombre d'identificateurs par programme compilé (Fig. 3.3.3.2.1. (c)).

Observations Nombre d'identificateurs	Fréquences	Fréquences cumulées
10	1/20	5/100
20	7/100	12/100
30	1/10	22/100
40	3/20	37/100
50	1/4	62/100
60	9/50	80/100
70	3/25	92/100
80	1/25	96/100
90	3/100	99/100
100	1/100	100/100

Fig. 3.3.3.2.1.(c) Tableau statistique

Moyenne = 49,5

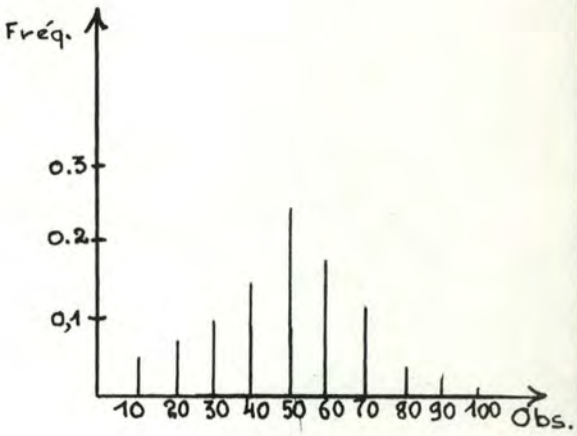


Fig. 3.3.3.2.1.(d) Diagramme en bâtonnets correspondant au tableau statistique Fig.3.3.3.2.1. (c)

A partir de ce tableau, combien d'entrées faut-il réserver pour la Table des Symboles ?

II. Stratégies possibles de réservation

En fait, deux stratégies sont concevables :

1) choisir une Table plutôt petite (par exemple 50 entrées)

. AVANTAGE : Cette solution présente l'avantage de minimiser la perte de place encourue, en moyenne, lors d'une compilation : seuls 37 % des programmes ne rempliront pas la Table des Symboles, et 22 % seulement n'utiliseront qu'au plus 3/5 de la place disponible.

. INCONVENIENT : Il faudra, 38 fois sur 100 compilations, appeler la procédure'TABLE PLEINE' et perdre de ce fait un temps considérable.

En effet, lorsque la Table de 50 entrées est remplie et qu'il reste des identificateurs à insérer, il tombe sous le sens qu'il faut agrandir cette Table (par exemple jusqu'à 100 entrées). Or, la fonction hash initiale (F_1) délivrait, par construction, des valeurs $\in \{x: 0 \leq x \leq 49\}$; elle ne convient donc plus dans ce nouveau contexte, et il importe de lui substituer une nouvelle fonction hash dont les valeurs $\in \{x : 0 \leq x \leq 99\}$.

Mais cette proposition implique que les identificateurs rangés par l'entremise de l'ancienne fonction hash le soient à nouveau par le truchement de la nouvelle fonction hash, puisque, désormais, c'est par elle qu'ils seront accédés .

Un exemple élémentaire fera mieux saisir la démarche.
Supposons que nous ayons une Table pleine de 5 entrées (Fig. 3.3.3.2.1. (e); nous désirons l'agrandir jusqu'à 9 entrées.
La fonction hash retenue est la division (1) et les enregistrements ont, pour notre commodité, des clés numériques.

---

(1) Cfr. p. 79

	CLE	VALEUR
0	45	'valeur'
1	66	'valeur'
2	37	'valeur'
3	78	'valeur'
4	29	'valeur'

Fig. 3.3.3.2.1.(e) Situation avant appel de la procédure 'Table Pleine'

	CLE	VALEUR
0	45	'valeur'
1	37	'valeur'
2	29	'valeur'
3	66	'valeur'
4		
5		
6	78	'valeur'
7		
8		

Fig. 3.3.3.2.1.(f) Situation à la sortie de la procédure 'Table Pleine'

On constate sur la figure Fig. 3.3.3.2.1.(f) que seul l'enregistrement de clé 45 conserve dans la Table agrandie le même numéro d'entrée que dans la Table initiale.

2) Choisir une Table plutôt grande (par exemple 70 entrées)
Naturellement, avantages et inconvénients sont inversés, c'est-à-dire :

. AVANTAGE : 4 % des programmes seulement auront recours à la procédure 'Table Pleine'.

. INCONVENIENT : la place perdue est, en moyenne, importante : 37 % des programmes utilisent au plus la 1/2 de la Table.

En général, cette seconde solution sera retenue. Mais il résulte que les méthodes hash qui résolvent des collisions sans chaînage acceptent une perte de place en moyenne non négligeable .
Nous n'y reviendrons plus, lorsque nous examinerons les performances des différentes variantes.

3.3.3.2.2. Essai linéaire (Linear probing)

1. Description générale

Dans cette perspective, historiquement la première, la fonction F_2 est ainsi définie :

$$F_2 : i \longrightarrow F_2(i) = i \text{ ou } [i * c] \text{ avec c constante}$$

C'est donc une fonction linéaire de i.

D'autre part, F_2 présente la propriété remarquable suivante :

" la différence entre deux entrées consécutivement examinées pour 1 clé de K quelconque est une constante ".

Démontrons le :

$$F_1(K) + F_2(i + 1) - [F_1(K) + F_2(i)] = F_2(i + 1) - F_2(i)$$

$$F_2(i + 1) - F_2(i) = i + 1 - i = 1$$

ou

$$(i + 1)\, c - ic = c$$

Dès lors, la procédure de calcul s'en trouve considérablement améliorée, puisque l'entrée suivante à investiguer est obtenue en incrémentant (modulo la taille logique de la Table) l'entrée actuelle de 1 ou de c .

2. Algorithme d'implémentation (1)

En fait, seuls les pas 1 et 3 de l'algorithme général (2) sont modifiés comme suit :

PAS	ALGORITHME	COMMENTAIRES
1	s := $F_1(K)$ q := s	En fait, la variable q n'est pas nécessaire, nous la maintenons uniquement pour ne pas recopier tout l'algorithme!
2	q := q + 1 [c] Si q $\geq$ M, q := q - M; aller en 2.	-

(1) KNUTH, Art of Programming, III, p. 519; HOPGOOD, Compiling Techniques, p. 23,

(2) Cfr. p. 99

3. Performances

La règle F_2 ainsi mise à jour est d'un calcul rapide (une simple décrémentation) et permet un parcours total de la Table.

Par ailleurs, sous réserve des conditions déjà exprimées (1), le nombre moyen d'enregistrements accédés

lors d'une recherche fructueuse

$$C_N \approx 1/2 \ (1 + \frac{1}{1-\alpha}) \qquad (2)$$

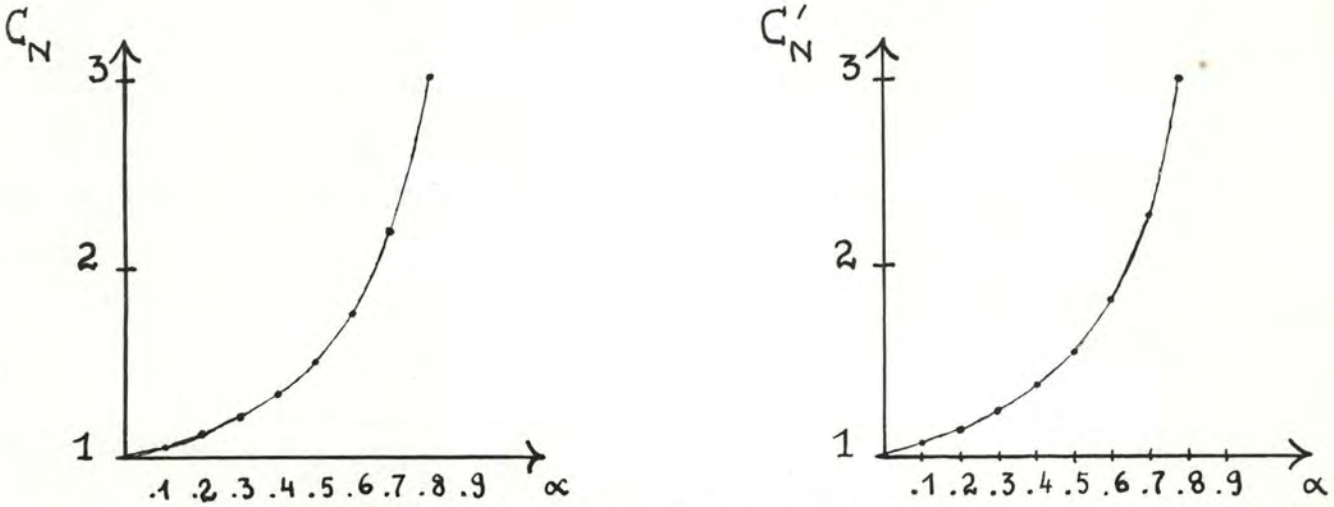
tandis que lors d'une recherche infructueuse

$$C'_N \approx 1/2 \ (1+ (\frac{1}{1-\alpha}) \ 2) \qquad (2)$$

Nous avons calculé, dans le tableau Fig. 3.3.3.2.2.(a), la valeur de ces expressions pour différentes valeurs de α et nous avons reporté les résultats sur les graphiques ci-après :

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
C_N	1.0556	1.125	1.214	1.333	1.500	1.750	2.167	3.000	5.500
C'_N	1.053	1.137	1.230	1.366	1.541	1.823	2.260	3.223	5.526

Fig. 3.3.3.2.2. (a) Tableau de calculs



Graphiques correspondant au tableau Fig. 3.3.3.2.2.(a)

---

(1) Cfr. p. 101.

(2) KNUTH, Art of Programming, III, p. 521.

4. Analyse des performances

Ces résultats font apparaître une dégradation indiscutable des performances, dès que le facteur de chargement α dépasse 0.7.
Toute amélioration éventuelle de la méthode implique donc, au préalable, de connaître la (ou les) cause(s) de ce soudain accroissement du nombre moyen d'accès lors d'une recherche (fructueuse ou infructueuse).
A cet effet, nous définirons d'abord la notion de groupements et montrerons ensuite comment les groupements demeurent responsables de la médiocrité des performances d'accès, lorsque α grandit.

I. Notion de groupements

a) Groupement primaire

On dit que la fonction F_2 engendre un groupement primaire si
étant donné 1) 2 clés K et J : $F_1(K) \neq F_1(J)$
2) $\exists$ i et j, $i \neq j$: $F_1(K) + F_2(i) = F_1(J) + F_2(j)$
alors :

$$F_1(K) + F_2(i) = F_1(J) + F_2(j) \Longrightarrow F_1(K) + F_2(i+k) = F_1(J) + F_2(j+k), \forall k \geq 1..n.$$

Concrètement, cela revient à dire que si, au cours de leur séquence d'essais, deux enregistrements de clé K et J, pour lesquelles la valeur initiale $F_1(K)$ et $F_1(J)$ est différente, aboutissent à la même entrée de la Table ($F_1(K) + F_2(i) = F_1(J) + F_2(j)$), alors le chemin qu'ils parcourent à partir de cet endroit là est identique.

b) Groupement secondaire

On dit que la fonction F_2 engendre un groupement secondaire si
étant donné 2 clés K et J : $F_1(K) = F_1(J)$
alors $F_1(K) = F(J) \Longrightarrow F_1(K) + F_2(i) = F_1(J) + F_2(i), \forall i \geq 1.$
En clair, la proposition signifie que si deux clés différentes sont telles que les valeurs initiales $F_1(K)$ et $F_1(J)$ sont identiques, tout le chemin qu'elles parcourent dans la Table est identique.

II. Influence des groupements sur les performances d'accès

Considérons, à titre d'exemple (1), la Table suivante de 19 entrées (Fig. 3.3.3.2.2.(b)).

9 d'entre elles sont occupées (en hachuré sur la figure).

La fonction F_2 est celle utilisée dans le cadre de la méthode dite par essai linéaire; c'est dire qu'elle engendre à la fois groupements primaire et secondaire.

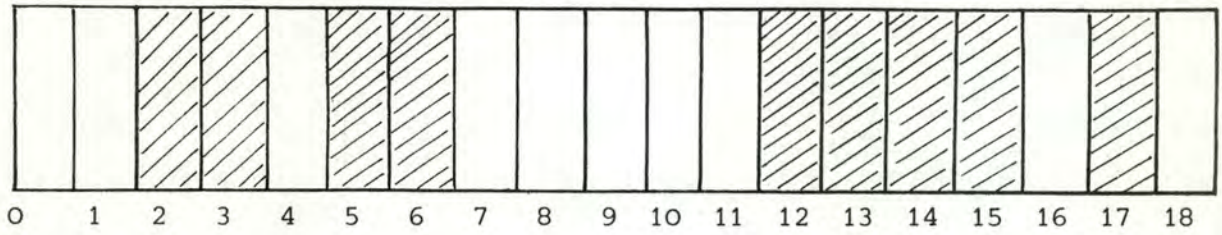


Fig. 3.3.3.2.2.(b) Situation typique d'une Table structurée selon la méthode dite par essai linéaire.

Le prochain enregistrement E_K de clé K à insérer viendra se loger dans un des espaces vides, mais tous ne sont pas équiprobables.

En effet, il sera inséré dans la position 16 si $12 \leq F_1(K) \leq 16$, tandis qu'il n'occupera la position 8 que si et seulement si $F_1(K) = 8$.

Donc, la position 16 est 5 fois plus probable que la position 8. Ce qui implique que les longs groupements (listes) tendent à devenir encore plus longs.

Et le problème se corse si une position vide entre deux groupements (par exemple, la position 4 ou 16) vient à se remplir; dans ce cas, en effet, deux groupements séparés se combinent.

Cette situation implique que si une clé K est telle, par exemple, que $F_1(K) = 12$, il lui faudra déjà accéder 5 fois à la Table lors de son insertion (et de toute recherche ultérieure); si $F_1(K) = 13$, 4 fois; si $F_1(K) = 14$, 3 fois, etc.

Or, nous venons de voir que plus α grandit, plus les groupements s'allongent.

Dès lors, plus α grandit, plus le nombre d'accès lors d'une insertion (et d'une recherche ultérieure) croît.

---

(1) KNUTH, Art of Programming, III, p. 520.

C'est pourquoi, les mathématiciens se sont efforcés de découvrir des fonctions F_2 qui évitent les groupements, et d'abord le groupement primaire.

3.3.3.2.3. Essai aléatoire (Random Probing)

1. Description générale

Les efforts de Morris (1) ont pour objectif de découvrir une fonction F_2 telle que :

1) l'image de F_2 soit $\{x: 1 \leq x \leq M-1\}$ si M représente la taille logique (nombre d'entrées) de la Table;
2) F_2 ne soit pas une fonction linéaire de i, autrement dit que la différence $F_2(i+1) - F_2(i)$ $\forall$ i $\in \{1 \ldots M-1\}$ ne soit pas une constante.

A cet effet, il propose la définition suivante de F_2 pour une Table dont la taille logique M est une puissance de 2 ($\exists$ n $\in$ N* M = 2^n):

$$F_2 : i \longrightarrow F_2(i) = \text{entier } (5^i \bmod 2^{n+2}/4)$$

On démontre théoriquement que l'image de F_2 est bien $\{x: 1 \leq x \leq M-1\}$.
Montrons-le sur un exemple.
Supposons que la Table(TABLE)compte 8 entrées (M = 2^3) numérotées de 0 à 7; l'ensemble des différentes valeurs de F_2 sera généré dans l'ordre suivant :

i	$5^i \bmod 2^5$	$F_2(i) = \text{entier } ([5^i \bmod 2^5]/4)$
1	5	1
2	25	6
3	29	7
4	17	4
5	21	5
6	9	2
7	13	3

Fig. 3.3.3.2.3.(a) Table des nombres générés par F_2.

(1) MORRIS, Scatter Storage.

Dès lors, si, au départ d'une entrée quelconque de la Table - soit la quatrième - nous voulons examiner TOUTES les entrées de la Table, il suffira de les inspecter dans l'ordre où elles apparaissent Fig. 3.3.3.2.3. (b).

Nb d'entrées déjà examinées i	p_i=entier($[5^i \bmod 2^5]/4$)	Numéro d'entrée examinée = (N° de l'entrée initiale + p_i) mod 8 (taille de la Table.
1	1	(4 + 1) mod 8 = 5
2	6	(4 + 6) mod 8 = 2
3	7	(4 + 7) mod 8 = 3
4	4	(4 + 4) mod 8 = 0
5	5	(4 + 5) mod 8 = 1
6	2	(4 + 2) mod 8 = 6
7	3	(4 + 3) mod 8 = 7

Fig. 3.3.3.2.3.(b) Séquence d'essais complète au départ de l'entrée 4 de la Table.

Manifestement, la Table est ainsi entièrement parcourue. Mais le groupement primaire n'est que PARTIELLEMENT éliminé. Le lecteur s'en convaincra aisément en considérant le cas suivant.

Supposons 2 clés L et K telles que

$$F_1(L) \neq F_1(K) \quad \text{et} \quad F_{22}(L) = F_{24}(K);$$

alors $F_{23}(L) = F_{25}(K)$, $F_{24}(L) = F_{26}(K)$, etc.

2. Algorithme d'Implémentation (2)

Quoique cet algorithme ne diffère pas, en substance, de l'algorithme général décrit (3) , les aménagements assez nombreux qu'il suppose nous incitent pourtant à le reproduire complètement.

Etant donné donc une table (TABLE) de M entrées ($\exists n \in \mathbb{N}^* : M = 2^n$), numérotées de 0 à 7, marquées '+' ou '-' selon qu'elles sont occupées ou vides, et comportant 2 autres champs au moins (CLE, où est rangée la clé

(1) On voit donc apparaître le groupement primaire.

(2) On trouvera dans MORRIS, Scatter Storage, p. ce programme écrit en FORTRAN.

(3) Cfr. p. 99

d'un enregistrement et VALEUR, où est rangée sa valeur), l'algorithme suivant structure l'accès à cette Table de la manière que nous venons de décrire.

PAS	ALGORITHME	COMMENTAIRES
1	R3 := F_1(K); R1 := 1; R4 :$_=$ R3	Phase d'initialisation comportant 1) le calcul de la fonction F_1 pour un argument K et le rangement de sa valeur dans le registre général R3. 2) l'initialisation à 1 du registre général R1 dont l'utilité apparaît au pas 3. 3) Le transfert de R3 dans R4, à la fin de préserver le contenu de R3 dont nous aurons ultérieurement besoin.
2	Si TABLE [R4] est 'vide', aller en 4; Sinon, si CLE [R4] = K, fin.	Cfr. commentaires de l'algorithme général, pp.99-100
3	R1 := R1 * 5 Si R1:=1, 'Table pleine'; Masquer tous les bits de R1 sauf les n + 2 de rang le plus bas; Placer le résultat dans R1; R2 := R1; Décaler R2 de 2 positions vers la droite;	Programme générateur (1). Test de fin de recherche; si R1 retrouve sa valeur initiale 1, alors la Table est pleine. Rappelons que la variable n représente la puissance de 2, telle que le nombre total d'entrées de la Table [M] = 2^n. On sauvegarde de la sorte le contenu de R1 ; à chaque nouvel essai, l'exponentiation 5^i est ainsi remplacée par une multiplication. Division par 4.

(1) Voir GRIES, Compiler Construction, p 218.

PAS	ALGORITHME	COMMENTAIRES
4	R4. := R3 + R2. Si R4 $\geq$ M, R4 = R4 - M; aller en 2.	Le registre de travail R4 recueille la somme R3 + R2.
5	TABLE [R4] est 'occupée' et CLE [R4] = K.	Procédure d'insertion. Cfr. algorithme général, p. 100.

3. Performances

La règle de résolution proposée par Morris s'avère relativement rapide ; le programme générateur (PAS 3), malgré les apparences, ne réclame, à chaque nouvel essai, guère plus de 6 instructions-machine. Néanmoins, parmi ces instructions se cache une multiplication dont on connaît l'effet ralentissant.

En outre, cette règle prend en charge toute la Table et, par rapport à la méthode dite par essai linéaire, les performances Temps-Logique se trouvent améliorées dans de notables proportions.

En effet, le nombre moyen d'accès lors d'une recherche fructueuse

$$\boxed{[C_N] \approx -\frac{1}{\alpha} \log (1-\alpha)}$$

tandis que le nombre moyen d'accès lors d'une recherche infructueuse

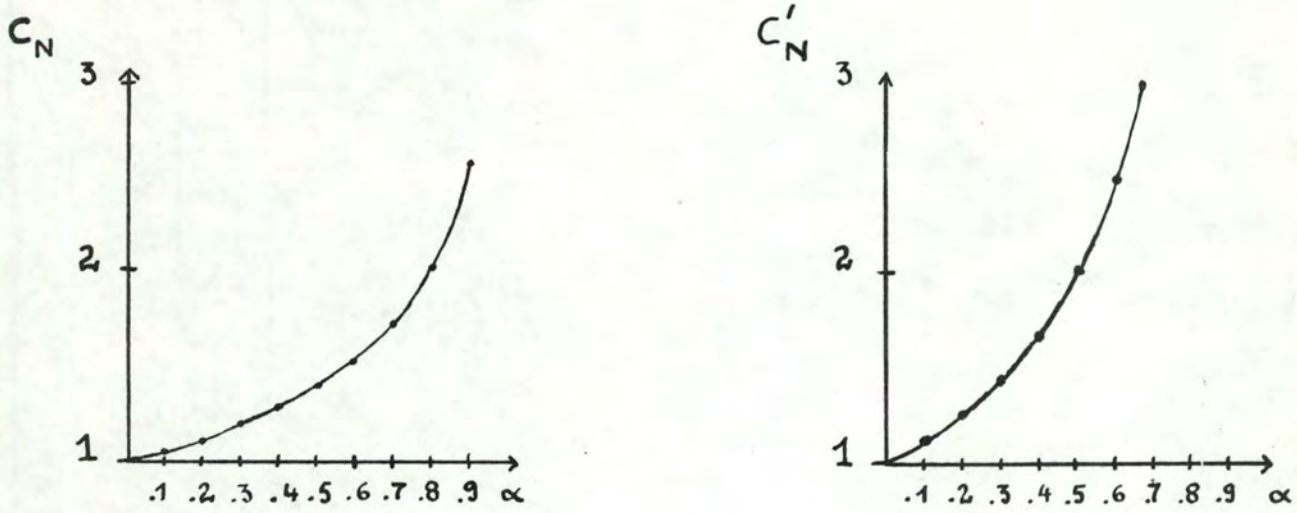
$$\boxed{[C'_N] \approx \frac{1}{1-\alpha}} \qquad (1)$$

Ces résultats sont calculés pour différentes valeurs de α dans le tableau Fig. 3.3.3.2.3.(c) et reportés sur les graphiques ci-après.

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
C_N	1.05	1.11	1.19	1.27	1.39	1.53	1.73	2.02	2.56
C'_N	1.11	1.25	1.42	1.66	2.	2.5	3.33	5.	10.

Fig. 3.3.3.2.3.(c) Tableau des Calculs

(1) MORRIS, Scatter Storage, p.40. Bell soutient que ces calculs ne tiennent aucun compte du groupement secondaire et propose les corrections selon lui adéquates. On se référera, à cet effet, à BELL, Quatratic Quotient, p. 109.



Graphiques correspondant au tableau des calculs Fig.3.3.3.2.3.(C).

3.3.3.2.4. Recherche Quadratique (Quadratic Search)

1. Description Générale

A) Sens général de la démarche

Le calcul de la fonction hash F_1, par la méthode de la division délivre donc, pour un argument K donné, une valeur entière k. L'idée de Maurer consiste alors, dans l'hypothèse où l'entrée k de la Table est occupée, d'"essayer" successivement les entrées $(k + ai^2 + bi) \bmod M$, la taille de la Table, $\forall\, i \in \{1, 2, 3 \ldots\}$.

En d'autres termes, F_2 est ainsi définie :

$$F_2 : i \longrightarrow ai^2 + bi$$

et

$$F_{2i}(K) = k + ai^2 + bi.$$

Naturellement a et b auront préalablement été fixés.

B) Obstacles apparents

Cette proposition se heurte, à première vue, à deux obstacles fort importants :

1) chaque nouvel essai réclame un calcul considérable :
   1 exponentiation, 2 multiplications et 2 additions.
   Du point de vue de la rapidité, l'idéal serait évidemment de se ramener à une simple décrémentation (incrémentation) (cfr. l'essai linéaire .).
2) a priori, nous ignorons si, de la sorte, la Table pourra être parcourue entièrement lors d'une recherche quelconque.
   Par bonheur, les propriétés des résidus quadratiques jettent quelque lumière sur notre problème.

   " Un résidu quadratique pour un nombre premier p est, par définition, un entier r ($0 < r < p$) pour lequel il existe i tel que $i^2 - r$ soit divisible par p. Il est, en outre, démontré que pour un nombre premier p et pour tous les entiers i, il n'existe que $(p - 1)/2$ résidus quadratiques." (1)

Concrètement, $\forall\, i \in N^*$ et $p \in \{x : x \text{ est premier}\}$, il n'existe que $(p - 1)/2$ restes différents de la division de i^2 par p.
Au mieux, nous parcourrons donc la moitié de la Table, si le nombre de ces entrées est premier.
Dès lors, lever ces deux obstacles revient à découvrir, pour un nombre premier p déterminé, des valeurs de a et de b telles que

1) à chaque essai, l'évaluation polynomiale puisse être remplacée par une simple incrémentation (décrémentation);
2) lors d'une recherche quelconque, $(p - 1)/2$ entrées différentes de la Table puissent être examinées;
3) que le groupement primaire soit éliminé, puisque c'est notre objectif avoué.

---

(1) RADKE, Quadratic Residue, p. 103.

C) Calcul de 'a' et de 'b'

Pour une table de M entrées (M premier), les valeurs de a et de b seront fixées de la manière suivante .

Calculons d'abord l'incrément nécessaire pour passer de l'essai n° i à l'essai n° i + 1

$$INC = a(i + 1)^2 + b(i + 1) - (ai^2 + bi) = \boxed{2\,ai + a + b}$$

Calculons ensuite la valeur initiale de l'incrément INC_0

$$INC_0 = \boxed{a + b}$$ (puisque initialement i vaut zéro).

Par ailleurs, cet incrément varie de valeur à chaque essai, en fonction de i, et l'incrément de l'incrément vaut

$$INCINC = 2\,a\,(i + 1) + a + b - \left[2\,a(i) + a + b\right] = \boxed{2\,a}$$

Le reste de la stratégie varie selon la richesse des instructions-machine.

Une possibilité décrite par Maurer consiste à :

1) décrémenter l'incrément de 1 à chaque essai

d'où

$$2\,a = -1$$
$$a = -\frac{1}{2}$$

et

2) donner pour valeur initiale à l'incrément, la valeur $(M - 1)/2$

d'où

$$a + b = (M - 1)/2$$
$$b = (M - 1)/2 - a$$
$$= (M - 1)/2 + \frac{1}{2}$$

On démontre que pour ces valeurs de a et de b, on peut parcourir exactement la moitié de la Table + 1.

Pour que le lecteur en soit convaincu, nous allons le montrer sur un exemple.

Soit une Table (TABLE) à 7 entrées numérotées de 0 à 6.

Imaginons que nous voulions calculer la séquence d'essais totale pour une clé K telle que $F_1(K) = 3$.

En initialisant a = 0.5 et b = 3.5, l'application de l'évaluation polynomiale donnera les résultats repris dans le tableau suivant :

i N° essai	ai^2	bi	$ai^2 + bi$	$(F_1(K) + ai^2 + bi) \bmod 7$	Entrée examinée
1	- 0.5	3.5	3	(3 + 3) mod 7 = 6	TABLE [6]
2	- 2	7	5	(3 + 5) mod 7 = 1	TABLE [1]
3	- 4.5	10.5	6	(3 + 6) mod 7 = 2	TABLE [2]
4	- 8	14	6	(3 + 6) mod 7 = 2	TABLE [2]
5	-12.5	17.5	5	(3 + 5) mod 7 = 1	TABLE [1]
6	-18	21	3	(3 + 3) mod 7 = 6	TABLE [6]

Nous montrerons, dans la présentation de l'algorithme, comment remplacer l'évaluation polynomiale par une simple incrémentation.

2. Algorithme d'Implémentation

Etant donné une table (TABLE) de M entrées (M premier) numérotées de 0 à M-1, chacune des entrées TABLE [i] contenant un champ indiquant si elle est vide ou non, un champ CLE [i] où est rangée la clé K de l'enregistrement E_K et un champ VALEUR [i] où est rangée la valeur associée à CLE [i], l'algorithme suivant construit la Table des Symboles selon la description générale que nous venons d'en faire.

PAS	ALGORITHME (1)	COMMENTAIRES
1	S := $F_1(K)$ R_1 = ((M-1)/2)+ 1	Calcul de la fonction F_1 par la méthode de la division. Initialisation du registre R1.
2	Si TABLE [S] est 'vide', aller en 4;	Cfr. l'algorithme général, p. 100

(1) Maurer se contente d'en donner le programme en langage de base pour des machines IBM 7094, UNIVAC 1108, SDS 940, IBM 360, in MAURER, Improved Hash, pp. 36 - 38.

OPERATIONS DE PRET ENREGISTREMENT D'UN EMPRUNT 29/03/83 17:23.12

COTE DU LIVRE : I18/8/17.....
LA TABLE DES SYMBOLES DU COMPILATEUR LESUISSE R. S
CLASSE-NUMERO UTILISATEUR : EX0537
BANZA MUTAMBAYI 0000 E R
EMPRUNT PENDANT LA PERIODE DU 29/03/83 AU 12/04/83
POUR TOUTE PROLONGATION, VOULEZ-VOUS VOUS PRESENTER ICI-MEME AVEC L'OUVRAGE.

PAS	ALGORITHME	COMMENTAIRES
	Sinon, si CLE [S] = K, recherche fructueuse.	
3	R1 := R1 - 1; Si R1 = O, 'Table pleine'; S := S + R1; aller en 2.	Test de fin de recherche et appel de la procédure 'Table Pleine' Préparation de l'essai suivant.
4	TABLE [S] est 'occupée' et CLE [S] = K.	Procédure d'insertion.

3. Performances

Sans nul doute, la règle de résolution des oollisions (F_2) ainsi découverte est d'un calcul rapide (5 instructions ASSEMBLER sur IBM 360). En termes d'accès, ses performances sont identiques à celles obtenues par Morris (1). Néanmoins, par rapport à la règle de résolution F_2 idéale, cette méthode présente deux désavantages :

1) Elle n'évite pas les groupements secondaires. A cet effet, Bell (2) propose une modification qui améliore de si peu les performances qu'il ne nous paraît pas opportun de nous y attarder.
2) Lors d'une recherche quelconque, elle ne prend en compte qu'une moitié de la Table. Selon Maurer (3), les statistiques ont prouvé que cela suffisait largement. Pour notre part, il nous paraît que le déclenchement de la procédure 'Table Pleine' (4) coûte trop cher pour qu'on prenne le risque de la mettre en action à mauvais escient, c'est-à-dire quand certaines entrées de la Table sont encore libres.

   C'est à réduire ce handicap que Radke s'est attaché.

---

(1) Cfr. p. 112.

(2) BELL, Quadratic Quotient.

(3) MAURER, Improved Hash, p. 36.

(4) Cfr. pp 103 - 104.

3.3.3.2.5. Recherche par la méthode des résidus quadratiques

1. Description générale

A) Sens général de la démarche

Quoique Maurer ait affirmé, sur la foi des statistiques, qu'il n'était point fâcheux, la plupart du temps, qu'une recherche complète ne prît en compte que la moitié des entrées de la Table, Radke (1) pense, néanmoins, qu'il serait préférable que, le cas échéant, toutes les entrées de la Table puissent être inspectées une et une seule fois, lors d'une séquence d'essais.

Il imagine donc de modifier l'idée de Maurer de la façon suivante : comme nous l'avons fait remarquer (2), le nombre d'entrées examinées (la 1/2 des entrées de la Table) par la méthode de recherche quadratique $(n + ai^2 \pm bi)$ ne dépend ni de a, ni de b, mais uniquement de i^2. En sorte que si nous décrétons que a = 1 et b = 0, nous parcourrons exactement le même nombre d'entrées.

Partant de cette constatation, Radke va chercher à découvrir 2 ensembles X et Y tels que pour un nombre premier p donné :

$X \cup Y = \{0, 1, \ldots p-1\} - n$ si n représente l'entrée examinée par l'application à la clé K de la fonction hash initiale (F_1).

$X \cap Y = \emptyset$

Les découvrir revient à formuler une règle qui permettra, à nouveau, de parcourir, s'il le faut, la Table entière lors d'une recherche.

B) Définition des ensembles X et Y

S'appuyant sur les travaux de Sierpinsky (3), Radke en vient à formuler le théorème suivant :

"Si la taille logique (nombre d'entrées) d'une table T est un

---

(1) RADKE, Quadratic Residue.

(2) Cfr. p. 114.

(3) SIERPINSKY, A Selection of Problems in the Theory of Numbers, Macmillan, N.J., 1964.

nombre premier p tel que $p > 2$ et $p = 4K + 3$, $\forall K \in N^*$; si la valeur n ($n < p$) désigne la valeur de la fonction hash initiale F_1 pour une clé donnée, alors les ensembles

$$X = \{(i^2 + n) \bmod p \mid i \in \{1,2, .. (p - 1)/2\}\}$$

et

$$Y = \{(p + 2n - (i^2 + n) \bmod p) \bmod p \mid i \in \{1,2, ... (p - 1)/2\}\}$$

ont les propriétés :

1) $X \cap Y = \emptyset$

2 $X \cup Y = \{i : i = 0, 1, 2 ..., (p - 1)\} - \{n\}$."

Il suffit alors d'écrire sous forme d'algorithme les propriétés découvertes et d'en montrer le fonctionnement sur un exemple.

2. Algorithme d'Implémentation

Etant donné une Table (TABLE) de M entrées ($M > 2, M = 4K + 3, \forall K \in N^*$ et M premier), de longueur fixe et numérotées TABLE [0] ... TABLE [M-1], marquées '+' ou '-' selon qu'elles sont occupées ou vides, l'algorithme suivant structure l'accès à la Table de la manière que nous venons de décrire.

PAS	ALGORITHME (1)	COMMENTAIRES
1	k := F_1 (S)	Calcul de la fonction hash initiale F_1 pour la clé S d'un enregistrement E_S; rangement de sa valeur dans la variable entière k.
	x := M + 2 k	Préparation du calcul de l'expression apparaissant au pas 4.
2	Si TABLE[K] est 'vide', aller en 6; Sinon, i := 1.	... premier champ marqué '-' Préparation du calcul de F_{21} (S).
3	Calculer R := $(i^2 + k) \bmod M$; Examiner Table [R]; Si TABLE [R] est 'vide'; aller en 6.	Application à la séquence d'essais du ième élément de l'ensemble X. ... et déclencher la procédure d'insertion.

(1) RADKE, Quadratic Residue, p. 104.

PAS	ALGORITHME	COMMENTAIRES
4	Calculer T:= $\{x - R\}$ mod p; Examiner TABLE [T]; Si TABLE [T] est 'vide', aller en 6;	Application à la séquence d'essais du ième élément de l'ensemble Y. ... et déclencher la procédure d'insertion.
5	i := i + 1; Si i $\leq$ (M - 1)/2, aller en 3; Sinon, OVERFLOW ...	Préparation de l'essai suivant. Test de fin de recherche ... et correction du test fautif de Radke. Appel de la procédure 'Table Pleine'.
6	Procédure d'insertion.	

Soit, à titre d'exemple, une Table (TABLE) à 7 entrées (M > 2, M = 4 k + 3 pour k = 1).
Supposons qu'en l'état où la Table se trouve Fig. 3.3.3.2.5.(a), la fonction hash initiale F_1 délivre pour un argument S (clé de l'enregistrement E_S) la valeur 5.
La séquence d'essais apparaît alors Fig. 3.3.3.2.5.(b).

	LIBRE	CLE	VALEUR
0	+	I	'valeur'
1	+	J	'valeur'
2	+	K	'valeur'
3	-		
4	+	L	'valeur'
5	+	M	'valeur'
6	+	N	'valeur'

Fig. 3.3.3.2.5.(a) Table avant la recherche de l'enregistrement E_S

Valeur de i	Pas de l'Algorithme appliqué	Entrée examinée	Résultat
-	1	5	Table [5] est 'occupée' M ≠ S
1	3	6	Table [6] est 'occupée' N ≠ S
1	4	4	Table [4] est 'occupée' L ≠ S
2	3	2	Table [2] est 'occupée' K ≠ S
2	4	1	Table [1] est 'occupée' J ≠ S
3	3	O	Table [O] est 'occupée' I ≠ S
(M - 1)/2= ⟸ 3	4	3	Table [3] est 'vide' Procédure d'insertion.

Fig. 3.3.3.2.5.(b) Séquence d'essais

La Table [TABLE] se présentera donc, après cette insertion, de la sorte:

	LIBRE	CLE	VALEUR
O	+	I	'valeur'
1	+	J	'valeur'
2	+	K	'valeur'
3	+	S	'valeur'
4	+	L	'valeur'
5	+	M	'valeur'
6	+	N	'valeur'

Fig. 3.3.3.2.5.(c) Situation de TABLE après l'insertion de l'enregistrement E_S de clé S.

3. Performances

La règle de résolution de Radke corrige, sans aucun doute, une faiblesse de la méthode quadratique due à Maurer, en ce qu'elle permet de parcourir toute la Table.
Cependant, elle présente le désavantage évident de réclamer une procédure de calcul plus longue.
En outre, les performances exprimées en nombre moyen d'accès n'ont pas encore, au dire même de l'auteur (1) été évaluées.
Jusqu'à plus ample information, on adoptera donc à l'égard de cette méthode une attitude réservée.

3.3.3.2.6. Méthode du double Hashing - Variante dite du Quotient Linéaire

1. Description générale

En fait, cette méthode ne diffère pas essentiellement de la méthode dite par essai linéaire (2), hormis - et c'est le point fondamental - que F_2 est ainsi définie:

$$F_2 : i \longrightarrow i * S(K)$$

où S est une fonction de la clé K INDEPENDANTE de la fonction hash initiale F_1.
Par exemple, Bell et Kaman (3) choisissent pour 1 argument K donné :

$F_1(K)$ = K mod M où M désigne la taille logique de la Table,
M $\in$ {nombres premiers} ;

S(K) = entier (K/M)

Autrement dit, la valeur de $F_1(K)$ égale le reste de la division de K par M, tandis que la valeur de S(K) en est le quotient (4).
De la sorte, la fonction F_2

1. reste linéaire, c'est-à-dire que $F_2(i + 1) - F_2(i)$ = Constante = S(K)
2. élimine quasi-totalement groupements primaire et secondaire.

---

(1) RADKE, Quadratic Residue, p. 105.

(2) Cfr. p. 105.

(3) BELL et KAMAN, Linear Quotient, pp. 675 - 676.

(4) L'"indépendance" de ces 2 fonctions est à entendre dans le sens que si 2 clés K et J sont telles que $F_1(K) = F_1(J)$, cela n'entraîne pas que S(K)=S(J). La remarque est de KNUTH, Art of Programming, III, p. 522.

2. Algorithme d'Implémentation

Sous les contraintes de l'algorithme général (1), l'algorithme suivant structure la Table des Symboles de la façon que nous venons de décrire.

PAS	ALGORITHME (2)	COMMENTAIRES
1	i := K mod M	Calcul de $F_1(K)$ et rangement de sa valeur dans la variable i.
2	Si TABLE [i] est 'vide', aller en 6; Sinon, si CLE [i] = K, la recherche est fructueuse.	Double comparaison entraînant les actions suivantes : 1) si l'entrée TABLE [i] est 'vide', déclenchement de la procédure d'insertion (pas 6) ; 2) si l'entrée TABLE [i] est 'occupée' et que clé [i] = K, recherche fructueuse; 3) sinon, passer en 3.
3	C := entier (K/M) + 1	Calcul de S(K) et rangement de sa valeur dans la variable C. On ajoute 1 pour éviter le cas où le quotient serait nul; car alors, on testerait indéfiniment la même entrée de la Table.
4	i := i + C ; si i ⩾ M, i := i - M	Préparation de l'essai suivant ... ... modulo la taille de la Table.
5	Si TABLE [i] est 'vide', aller en 6; Sinon, si CLE [i] = K, l'algorithme se termine avec succès; sinon, aller en 4.	Idem qu'au pas 2.

(1) Cfr. p. 99

(2) Aménagement de l'algorithme de BELL et KAMAN, Linear Quotient, p. 676, dans notre cadre habituel.

PAS	ALGORITHME	COMMENTAIRES
6	Si N = M - 1, l'algorithme se termine avec OVERFLOW; sinon, N = N + 1, TABLE [i] est 'occupée' et CLE [i] = K.	Procédure d'insertion. Cfr. les commentaires de l'algorithme général (pas 4), p. 100.

3. Performances

La méthode du Double Hasking (variante dite du Quotient Linéaire)

1. permet, à partir d'une entrée quelconque, un parcours complet de la Table, puisque la taille logique M de la Table est, par hypothèse, un nombre premier.
   Nous sommes donc renvoyés au cas examiné p. 106
2. est d'un calcul rapide : une division suivie d'incrémentations
3. évite, pour l'essentiel, groupements primaire et secondaire.

Enfin, le nombre moyen d'enregistrements accédés lors d'une recherche fructueuse

$$C_N \sim + \frac{1}{\alpha} \log (1/1 - \alpha) \qquad (1)$$

tandis que le nombre moyen d'enregistrements accédés lors d'une recherche infructueuse

$$C'_N \sim 1/1 - \alpha \qquad (1)$$

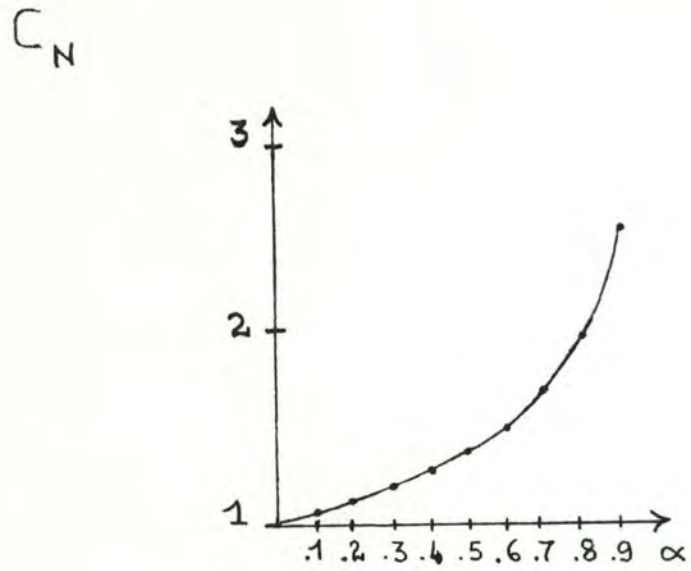
Les valeurs de C_N sont calculées pour différentes valeurs de α dans le tableau Fig. 3.3.3.2.6.(a) et reportées sur le graphique qui suit (2).

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
C_N	1.05	1.11	1.19	1.27	1.38	1.52	1.72	2.01	2.55

Fig. 3.3.3.2.6.(a) Calcul des valeurs de C_N

(1) BELL et KAMAN, Linear Quotient, pp. 676 - 677. Les expériences effectuées dans le cadre d'un compilateur COBOL implémenté sur une machine PDP-10 ont, semble-t-il, corroboré ces résultats.

(2) Les valeurs de C'_N sont identiques à celles obtenues par Morris et Maurer. Le lecteur voudra bien se référer p. 112.



QUOTIENT LINÉAIRE - RECHERCHE FRUCTUEUSE.

3.3.3.2.7. Méthode du Double Hashing - Optimisation de Brent

1. Description générale

Dans son principe, l'idée de Brent (1) est indépendante de toute méthode de résolution des collisions, et seules ses conditions d'application la lient actuellement à la méthode de résolution dite du Double Hashing.

C'est pourquoi, nous envisagerons d'abord le sens général de la démarche; nous donnerons ensuite un exemple qui la fera saisir dans son détail; nous énoncerons enfin ses conditions d'application.

A) Sens général de la démarche

Dans la plupart des programmes compilés, il s'avère (2) que le nombre de recherches fructueuses des clés l'emporte, et de beaucoup, sur celui des insertions.

(En fait, ces deux nombres seraient égaux si, et seulement si, chaque clé - ou identificateur - n'était référencé qu'une seule fois).

---

(1) BRENT, Reducing Time.

(2) Voir les statistiques portant sur un programme COBOL, où la proportion est de 14 à 1 in BELL et KAMAN, Linear Quotient, pp. 675 - 677.

Aussi, Brent propose-t-il d'optimiser (diminuer) si possible le temps moyen d'une recherche fructueuse, en effectuant davantage de travail au moment d'une insertion.
Quel travail effectuer ? Essentiellement bouger des enregistrements.
Dans quelles conditions et selon quelle méthode ? Un exemple, pensons-nous, illustrera mieux le procédé que n'importe quelle considération théorique.

B) Exemple du procédé

Considérons la table suivante (Fig. 3.3.3.2.7.(a)) dans laquelle sont déjà rangés les enregistrements E_K, E_L, E_M, E_N de clé respective K, L, M, N.
Supposons que par l'entremise d'une fonction hash F_1, et d'une fonction de résolution des collisions F_2, nous voulions effectuer une "recherche" de l'enregistrement E_A de clé A.

	LIBRE	CLE	VALEUR
0	-		
1	+	K	'valeur'
2	-		
3	+	L	'valeur'
4	+	M	'valeur'
5	-		
6	+	N	'valeur'

Fig. 3.3.3.2.7(a) Etat de la Table avant la recherche de l'enregistrement E_A de clé A.

Soit $F_1(A) = 3$. Une collision apparaît, qu'il faut résoudre selon la procédure générale (1).
Par malchance, la séquence d'essais successifs, obtenue alors en utilisant la fonction F_2, permet l'examen - en vain - des entrées 4 et 6, avant de trouver enfin une entrée vide, soit la deuxième.

(1) Cfr. pp 98 sq.

$F_1(A) + F_2(i)$	Valeur	Entrée examinée	Résultat
$F_{20}(A)$	3	TABLE [3]	'occupée' et L ≠ A
$F_{21}(A)$	4	TABLE [4]	'occupée' et M ≠ A
$F_{22}(A)$	6	TABLE [6]	'occupée' et N ≠ A
$F_{23}(A)$	2	TABLE [2]	Libre

Fig. 3.3.3.2.7.(b) Séquence d'essais dans le cadre d'une recherche de l'enregistrement E_A de clé A.

L'action à entreprendre va naturellement différer suivant qu'on se place dans l'optique qui fut traditionnellement nôtre ou dans l'optique nouvelle de Brent.

α/ Optique Traditionnelle

L'enregistrement E_A de clé A est inséré dans TABLE [2] et chaque fois qu'il faudra l'atteindre, il importera d'accéder (4) fois à la Table des Symboles.

	LIBRE	CLE	VALEUR
0	-		
1	+	K	'valeur'
2	+	A	'valeur'
3	+	L	'valeur'
4	+	M	'valeur'
5	-		
6	+	N	'valeur'

Fig. 3.3.3.2.7.(c) Etat de la Table après l'insertion de l'enregistrement E_A dans l'optique traditionnelle.

β/ Optique de Brent

Avant d'insérer l'enregistre E_A de clé A :

1) Dressons la liste des enregistrements accédés en vain par la règle de résolution F_{2i} appliquée à l'identificateur A.
   Cette liste contiendra en l'occurrence :
   - E_L de clé L
   - E_M de clé M
   - E_N de clé N.

2) Pour chacun de ces enregistrements, tâchons de connaître le numéro de la prochaine entrée libre que l'on rencontrerait en ajoutant la valeur de F_2 autant de fois qu'il le faudra à leur numéro d'entrée actuel.
   Imaginons, par exemple, qu'en ajoutant 1 fois la valeur de F_2 au numéro d'entrée 3 de l'enregistrement E_L, nous trouvions le numéro d'entrée 5 . Table [5] est vide.
   Dès lors, nous sommes en mesure d'optimiser une recherche ultérieure de l'enregistrement E_A de clé A, en procédant comme suit :
   1) Transférons l'enregistrement E_L de clé L de la 3e à la 5e entrée. De la sorte, nous allongeons d'1 accès à la Table chaque recherche de E_L. Par contre, ce transfert libère l'entrée n° 3.
   2) Insérons l'enregistrement E_A de clé A dans TABLE 3
      Nous diminuons donc chacune de ces recherches de 4 - 1 = 3 accès (1).

   Le gain total pour une recherche de E_A et de E_L est donc de 3 (gain de E_A) - 1 (perte de E_2) = 2 accès.
   Et au sortir de cette procédure, la Table aura donc la nouvelle configuration que voici :

---

(1) L'enregistrement E_A rangé dans TABLE [3] sera dorénavant accessible en un seul accès, puisque nous avons supposé que F_1 (A) = 3.

	LIBRE	CLE	VALEUR
0	-		
1	+	K	'valeur'
2	-		
3	+	A	'valeur'
4	+	M	'valeur'
5	+	L	'valeur'
6	+	N	'valeur'

Fig. 3.3.3.2.7.(d) Etat de la Table après l'insertion de E_A, selon l'optique de Brent

Naturellement, l'opération doit se terminer par un gain, sans quoi on procédera selon la manière traditionnelle.
Cet ingénieux stratagème exige, malheureusement, pour son application des conditions qui restreignent sévèrement son champ d'action.

c) Conditions d'application

Ces conditions ont essentiellement trait aux caractéristiques de la Fonction F_2 et peuvent s'énoncer de la sorte :

1) F_2 sera une fonction linéaire de i (nombre d'entrées déjà examinées). Cette condition est pratiquement nécessaire.
En effet, lorsqu'il s'est agi de calculer le n° d'entrée suivante accédée par l'enregistrement E_L de clé L (1), nous avons utilisé la fonction F_2 sans nous préoccuper de l'itinéraire suivi en son temps pour insérer l'enregistrement E_L de clé L. Ceci n'est possible que si F_2 est linéaire car, dans cette hypothèse, $F_2(i+1) - F_2(i) = \text{Constante}$, $\forall i \in \{1 \ldots p-1\}$.
Autrement dit, le numéro de l'entrée suivante est connu dès qu'on connaît le numéro de l'entrée actuelle et la constante.
Cette condition revient à exclure les méthodes de résolution des collisions dues à Morris, Maurer et Radke.

(1) Cfr. p. 128.

2) F_2 évitera groupements primaire et secondaire, et singulièrement le groupement primaire.

En effet, par définition, ce type de groupement apparaît lorsque F_2 est telle que

si au cours de leur séquence d'essais, 2 enregistrements de clé différente - soit L et M -, pour lesquelles la valeur de la fonction hash initiale est différente - $F_1(L) \neq F_1(M)$ - aboutissent à la même entrée de la Table, le chemin qu'ils parcourent à partir de cet endroit-là est identique. (1)

Choisir F_2 ainsi caractérisée équivaut donc, dans la procédure d'optimisation, à effectuer un travail parfaitement inutile, puisque, dans ce cas, le gain sera nul, par définition.

Ces deux conditions entraînent que seule des méthodes connues, la méthode dite du Double Hashing y satisfait présentement. Voilà donc pourquoi l'idée de Brent se développe dans le cadre de cette méthode.

2. Algorithme d'Implémentation

Sous les contraintes habituelles maintes fois répétées (2), l'algorithme suivant dû à Brent structure la Table des Symboles de la façon que nous venons de décrire. Soit à rechercher l'enregistrement E_K de clé K

PAS	ALGORITHME (3)	COMMENTAIRES
1	R := $F_1(K)$ = K mod p;	Calcul de la fonction hash initiale selon la méthode de la division, rangement de sa valeur dans la variable de travail R.
	Q := S(K) = (K mod (p-2)) + 1	Calcul de la règle de résolution selon la méthode de la division (diviseur p - 2) et rangement de sa valeur dans la variable de travail Q.

(1) Cfr. p. 107.

(2) Cfr. p. 99.

(3) Nous resterons à un niveau assez logique. Le lecteur en trouvera le programme écrit en FORTRAN (57 instructions) dans BRENT, Reducing Time, p. 109.

PAS	ALGORITHME	COMMENTAIRES
	T := R ;	Initialisation de la variable T qui nous permettra de sauvegarder, pour un emploi ultérieur, les variables R et Q.
2	Si TABLE [T] est 'vide', insérer l'enregistrement E_K de clé K; Sinon, si CLE [T] = K, recherche fructueuse.	Idem algorithme général (1). Aucun gain n'est possible en l'occurrence, puisque une recherche de l'enregistrement E_K de clé K ne nécessitera qu'un seul accès.
3	T := T + Q ; Si T ⩾ M; T = T - M ;	Préparation de l'essai suivant mod M, la taille logique de la Table .
4	Si Table [T] est 'vide', aller en 5; Si CLE [T] = K, la recherche est fructueuse; Si CLE [T] = CLE [R] , procédure 'Table pleine'. Sinon, aller en 3.	... et entamer la procédure d'optimisation. Nous sommes en effet revenus à notre point de départ.
5	S := T ; Considérons à présent, l'ensemble Z des entrées qui viennent d'être examinées : $Z = \{T_0, T_1, \ldots T_n\}$ où T_0 désigne l'entrée de la Table inspectée en appliquant zéro fois la règle de résolution F_2 etc.	Rangement de la valeur finale T dans la variable S. Parce que nous avons sauvé la valeur de $F_1(K)$ et de S(K) respectivement dans R et Q, nous pouvons aisément reconstituer cet ensemble. $T_0 = R$ $T_1 = R + Q$ $T_2 = R + 2\,Q \ldots$

(1) Cfr. p. 100.

PAS	ALGORITHME	COMMENTAIRES
	Pour chacune de ces entrées,	
	calculer Q_i := S(CLE $[T_i]$)	où Q_i est, par hypothèse, indépendante de F_1 (CLE $[T_i]$) ;
	Pour $j \geq 1$, $i + j < n$ et jusqu'à trouver une entrée libre, essayer l'entrée TABLE $T_i + jQ_i$	La condition $i + j < n$ exprime en fait la condition nécessaire pour qu'il y ait gain. En effet, l'indice i exprime le nombre de fois que la fonction F_2 a été, à ce moment-ci, appliquée à la clé K, tandis que l'indice j désigne le nombre de fois que la fonction F_2 a été appliquée à CLE $[T_i]$. Comme nous savons qu'au total la fonction F_2 sera appliquée n fois à la clé K pour trouver une place libre, nous optimiserons la procédure <u>si et seulement si</u> $i + j < n$
	Noter les coordonnées de la première entrée libre ainsi trouvée, c'est-à-dire : i, j, Q_i T_i $i + j$ et les ajouter à la liste des candidats à l'optimisation.	... étant donné les contraintes $(i+j< n)$, il se peut que pour certains T_i, nous ne trouvions aucune entrée libre ! Ces coordonnées vont nous permettre ultérieurement : 1) de choisir le gain maximum, 2) d'agir en conséquence.
	Au terme de cette boucle, <u>2 solutions</u> sont concevables : 1) la liste des candidats à	

PAS	ALGORITHME	COMMENTAIRES
	l'optimisation est <u>vide</u>; alors TABLE [S] = 'occupée'; CLE [S] = K VALEUR [S] = 'valeur de E_K'	... L'entrée trouvée par le pas 4 est la meilleure possible et nous avions mémorisé son numéro dans la variable S.
	2) la liste des candidats contient <u>une ou plusieurs entrées</u> (T_i); alors	
	. choisir celle pour laquelle i + j est le plus petit (Dans l'éventualité d'une égalité, choisir celle pour laquelle i est le plus petit).	... celle pour laquelle le gain est maximum.
	. procéder aux actions suivantes :	
	TABLE [$T_i + jQ_i$] = 'occupé' CLE [$T_i + jQ_i$] = CLE [T_i] VALEUR [$T_i + jQ_i$] = VALEUR [T_i] CLE [T_i] = K VALEUR [T_i] = 'valeur de E_K'	... Transferts qui optimisent la procédure de recherche.

Illustrons <u>par un exemple</u> cette démarche à première vue complexe. Considérons la Table (TABLE) à 7 entrées numérotées de 0 à 6 (Fig. 3.3.3.2.7.(e)).

Pour notre commodité, les enregistrements qu'elle contient ont des clés numériques.

Soit à "rechercher" dans cette Table l'enregistrement de clé 38.

	LIBRE	CLE	VALEUR
0	+	49	'valeur'
1	+	22	'valeur'
2	-		
3	+	24	'valeur'
4	+	46	'valeur'
5	+	26	
6	-		

Fig. 3.3.3.2.7(e) Situation initiale de la Table.

PAS 1

R : = F_1 (38) = 38 mod 7 = 3

Q : = F_2 (38) =(38 mod 5) + 1 = 4

T : = 3

PAS 2

TABLE [T] est 'occupée' et CLE [3] ≠ 38; passer en 3.

PAS 3 et 4

Applications successives de F_2 jusqu'à trouver une entrée occupée de clé 38 ou une entrée libre.

i Nombre d'entrées déjà examinées	Valeur de la Variable T	Entrée examinée	Résultat
1	(3 + 4) mod 7= 0	TABLE [0]	'occupé' et CLE 0 ≠ 38
2	(3+2.4) mod 7= 4	TABLE [4]	'occupé' et CLE 4 ≠ 38
3	(3+3.4) mod 7= 1	TABLE [1]	'occupé' et CLE 1 ≠ 38
4	(3+4.4) mod 7= 5	TABLE [5]	'occupé' et CLE 5 ≠ 38
5	(3+5.5) mod 7= 2	TABLE [2]	'libre' l'enregistrement de clé 38 n'est pas dans la Table. Il faut donc l'y insérer, mais auparavant ...

PAS 5 : Procédure d'optimisation

L'ensemble Z initial se présente de la sorte :

$\{T_0 = 3, T_1 = 0, T_2 = 4, T_3 = 1, T_4 = 5, T_5 = 2\}$.

L'indice le plus élevé de T_i = 5

A) i = 0, $T_0 = 3$, CLE $[T_0] = 24$, Q_0 = S (CLE $[T_0]$) = 5

Numéro de l'essai (j)	Valeur de i + j	i + j = 5 ?	Valeur de T T:= $(T_i + j\,Q_i)$ mod 7	Entrée examinée	Résultat
1	1	1 < 5	1	TABLE [1]	'occupé'
2	2	2 < 5	6	TABLE [6]	'libre' et 'sortie'

Conclusion

Nous avons découvert, dans la séquence d'essais parcourue pour insérer l'enregistrement de clé 38, une première entrée ($T_0 = 3$) qui permettrait d'optimiser la procédure de recherche.

Notons, dans la liste des candidats, les coordonnées de T_0.

A la sortie, cette liste a donc l'aspect suivant :

i	j	Q_i	T_i	i + j
0	2	5	3	2

Fig. 3.3.3.2.7.(f) Liste des candidats à la sortie A

B) i = 1, T_1 = 0, CLE $[T_1] = 49$, Q_1 = S (CLE $[T_1]$) = 5

Numéro de l'essai (j)	Valeur de i + j	i + j = 5 ?	Valeur de T T:= $(T_i + j\,Q_i)$ mod 7	Entrée examinée	Résultat
1	2	2 < 5	5	TABLE [5]	'occupée'
2	3	3 < 5	3	TABLE [3]	'occupée'
3	4	4 < 5	1	TABLE [1]	'occupée'
4	5	5 = 5	-	-	'sortie'

Conclusion

La liste des candidats n'est pas modifiée.

C) i = 2, T_2 = 4, CLE $[T_2]$ = 46, Q_2= 5 (CLE $[T_2]$) = 2

Numéro de l'essai (j)	Valeur de i + j	i + j= 5 ?	Valeur de T T:=(T_{i+j} Q_i) mod 7	Entrée examinée	Résultat
1	3	3 < 5	6	TABLE [6]	'libre' et 'sortie'

Conclusion

Voici un second candidat possible que nous allons ajouter à la liste déja existante. Elle se présentera dès lors de la sorte :

i	j	Q_i	T_i	i + j
0	2	5	3	2
2	1	4	4	3

Fig. 3.3.3.2.7 (g) Liste des candidats à la sortie C)

D) i = 3, T_3 = 1, CLE $[T_3]$= 22, Q_3 = S (CLE $[T_3]$) = 3

Numéro de l'essai (j)	Valeur de i + j	i + j = 5 ?	Valeur de T T:= (T_{i+j} Q_i) mod 7	Entrée examinée	Résultat
1	4	4 < 5	4	TABLE [4]	'occupée'
2	5	5 = 5	-	-	'sortie'

Conclusion

La liste des candidats reste identique à ce qu'elle était avant d'entrer dans D.

E) Les entrées T_4 (i = 4) et T_5 (i = 5) ne sont pas à prendre en considération puisque d'emblée i + j $\geq$ 5.

Conclusion

La liste finale descandidats est donc décrite Fig. 3.3.3.2.7 (g)

Conclusion du PAS 5

Deux entrées de la séquence d'essais (T_0 = 3 et T_2 = 4) permettent donc d'optimiser dans le sens souhaité la procédure de recherche. Laquelle choisir ? L'algorithme répond :

" Comparer les sommes i + j et prendre la plus petite, c'est-à-dire celle pour laquelle le gain est maximum."

En conséquence, nos hésitations s'évanouissent et les actions suivantes seront entreprises :

TABLE [6] = TABLE [3]
TABLE [3] = enregistrement de clé 38.

A la sortie de la procédure la Table (TABLE) aura donc ce nouvel aspect :

	LIBRE	CLE	VALEUR
0	+	49	'valeur'
1	+	22	'valeur'
2	-		
3	+	38	'valeur'
4	+	46	'valeur'
5	+	26	'valeur'
6	+	24	'valeur'

Fig. 3.3.3.2.7.(h) Table à la sortie de la procédure d'optimisation.

3. Performances

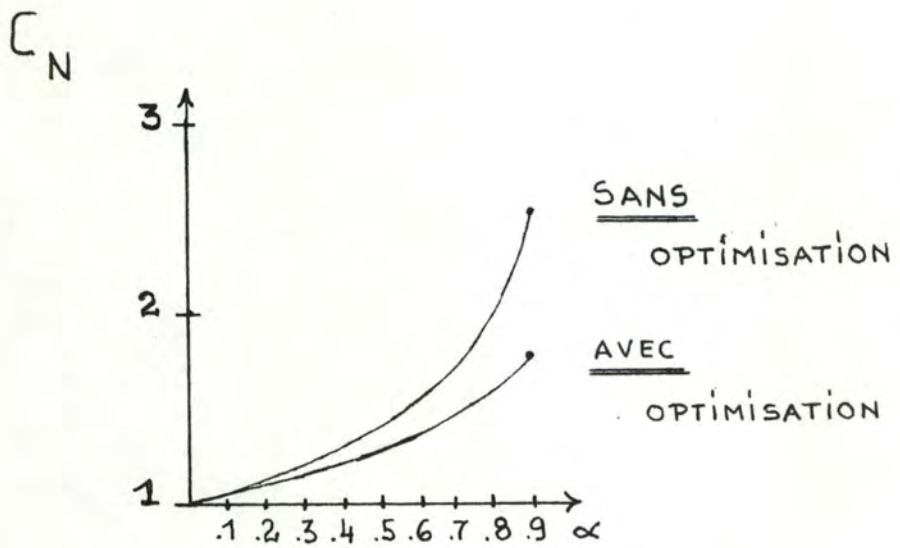
Puisque tout le travail de Brent vise à améliorer le nombre moyen d'accès à la Table lors d'une recherche fructueuse (C_N), seule cette valeur est susceptible de changement et, selon les calculs de l'auteur, sa nouvelle valeur devrait être proche de

$$C_N \approx 1 + \alpha/2 + \alpha^3/4 + \alpha^4/15 - \alpha^5/18 + 2\alpha^6/15 \ldots \quad (1)$$

Nous avons calculé dans le tableau Fig. 3.3.3.2.7(i) la valeur de C_N pour différentes valeurs de α et les avons reportées sur le graphique suivant, où figurent, à titre de comparaison, les valeurs de C_N obtenues par la méthode de résolution par Double Hashing sans optimisation (Méthode dite du Quotient Linéaire) (2).

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
C_N	1.05	1.10	1.15	1.21	1.28	1.36	1.45	1.60	1.80

Fig. 3.3.3.2.7.(i)



(1) BRENT, Reducing Time, pp. 106 - 107.

(2) Cfr. pp 124-125.

En conclusion, la procédure d'optimisation de Brent semble donc apporter une amélioration sensible de la performance C_N.
Cependant, il convient de garder à l'esprit que cette amélioration est le fruit d'un travail dont il importe de mesurer l'importance.

Brent se livre donc à des expérimentations, et en conclut que :

> "le gain total de temps d'accès à la Table reste considérable lorsque la Table tend à se remplir et que chaque entrée est recherchée plusieurs fois." (1)

Et, précisément, cette dernière condition est satisfaite par hypothèse.

3.3.3.2.8. Conclusion sur les méthodes de résolution des collisions sans chaînage

Peut-on comparer, valablement, entre elles les performances de ces diverses méthodes ?
Sans doute le problème ne présente-t-il aucune difficulté du point de vue de la PLACE-MEMOIRE qu'elles postulent : nous avons vu que cette place était pour toutes équivalente . (2)
L'évaluation de leur TEMPS REEL DE RECHERCHE d'un enregistrement se révèle, en revanche, beaucoup plus complexe.
Nous avons écrit en effet que le rapport temps logique de recherche - temps réel n'était plus immédiat. (3)
Avant tout, il nous fallait donc connaître, outre le temps logique de recherche (nombre d'accès à la Table), tous les autres facteurs capables d'influencer le temps réel de recherche. Notre étude nous en a fait découvrir deux :

1) le temps de calcul de la fonction F_2
2) le temps moyen perdu à réorganiser la Table 'Table Pleine', alors qu'elle n'est pas réellement pleine; en d'autres termes, le temps moyen perdu si la fonction F_2 ne couvre pas l'ensemble des entrées de la Table (par exemple, la méthode de recherche quadratique de Maurer).

Nous allons donc examiner, pour chacun des trois facteurs dont dépend le temps réel de recherche, les performances des différentes métho-

(1) BRENT, Reducing Time, pp. 108 - 109.
(2) Cfr. p. 104.
(3) Cfr. p. 101.

des de résolution des collisions sans chaînage.

1. Calcul de la fonction F_2

Naturellement, seuls sont pris en compte les éléments qui interviennent directement dans le calcul de F_2. Par ailleurs, conformément à notre méthode générale, nous nous en tiendrons à un niveau assez logique.

Néanmoins, le tableau comparatif ci-après donne une idée grossière du Temps global réel de calcul : quelque riche que soit l'ensemble des instructions de base d'une machine actuelle, il est clair, par exemple, que la méthode dite par essai linéaire réclame moins de temps de calcul que la méthode dite par essai aléatoire.

ESSAI LINEAIRE	ESSAI ALEATOIRE (1)	RECHERCHE QUADRATIQUE (MAURER)	RECHERCHE QUADRATIQUE (RADKE)	DOUBLE HASHING (BRENT)
1 addition	1 multiplication 1 exponentiation 1 modulo 1 division par une puissance de 2 1 comparaison (2)	1 soustraction 1 comparaison (2)	1 addition et 1 multiplication initiales 2 additions 2 modulo 1 soustraction	1 division initiale 1 addition

Conclusion : Les méthodes de recherche linéaire et de recherche quadratique (Maurer) obtiennent de ce point de vue d'excellents résultats; elles sont suivies de près par la méthode du Quotient Linéaire.

---

(1) Selon la logique du programme FORTRAN écrit par MORRIS, Scatter Storage, p.

(2) Ces méthodes introduisent dans la procédure de calcul le test final ...

2. Temps moyen perdu, si F_2 ne couvre pas toute la Table.

Il est difficile à évaluer théoriquement et statistiquement. Cependant, il ressort de notre exposé qu'en tout état de cause, seule la méthode de recherche quadratique (Maurer) se trouve, à cet égard, pénalisée.

3. Temps logique de recherche

Nous nous bornerons à reproduire sur des graphiques comparatifs (Fig. 3.3.3.2.8 (a) et 3.3.3.2.8. (b)) le nombre moyen d'enregistrements accédés que chacune de ces méthodes réclame pour une recherche fructueuse ou infructueuse.
Pour le détail, le lecteur voudra bien consulter la section 3.3.3.3.2.

Conclusion

Puisque la recherche fructueuse l'emporte - et de loin (1) - en fréquence sur le nombre de recherches infructueuses, nous attribuerons à la méthode du Double Hashing (optimisation de Brent) les meilleures performances Temps-Logique de recherche, même si, ne l'oublions pas (2), elle réclame un travail supplémentaire au moment de l'insertion d'un nouvel enregistrement dans la Table.
Au total, il apparaît que parmi toutes les méthodes de résolution sans chaînage, la méthode du Double Hashing (optimisation de Brent) obtient les meilleurs résultats globaux.

---

(1) Cfr. p. 125.

(2) Cfr. p. 131. PAS 5.

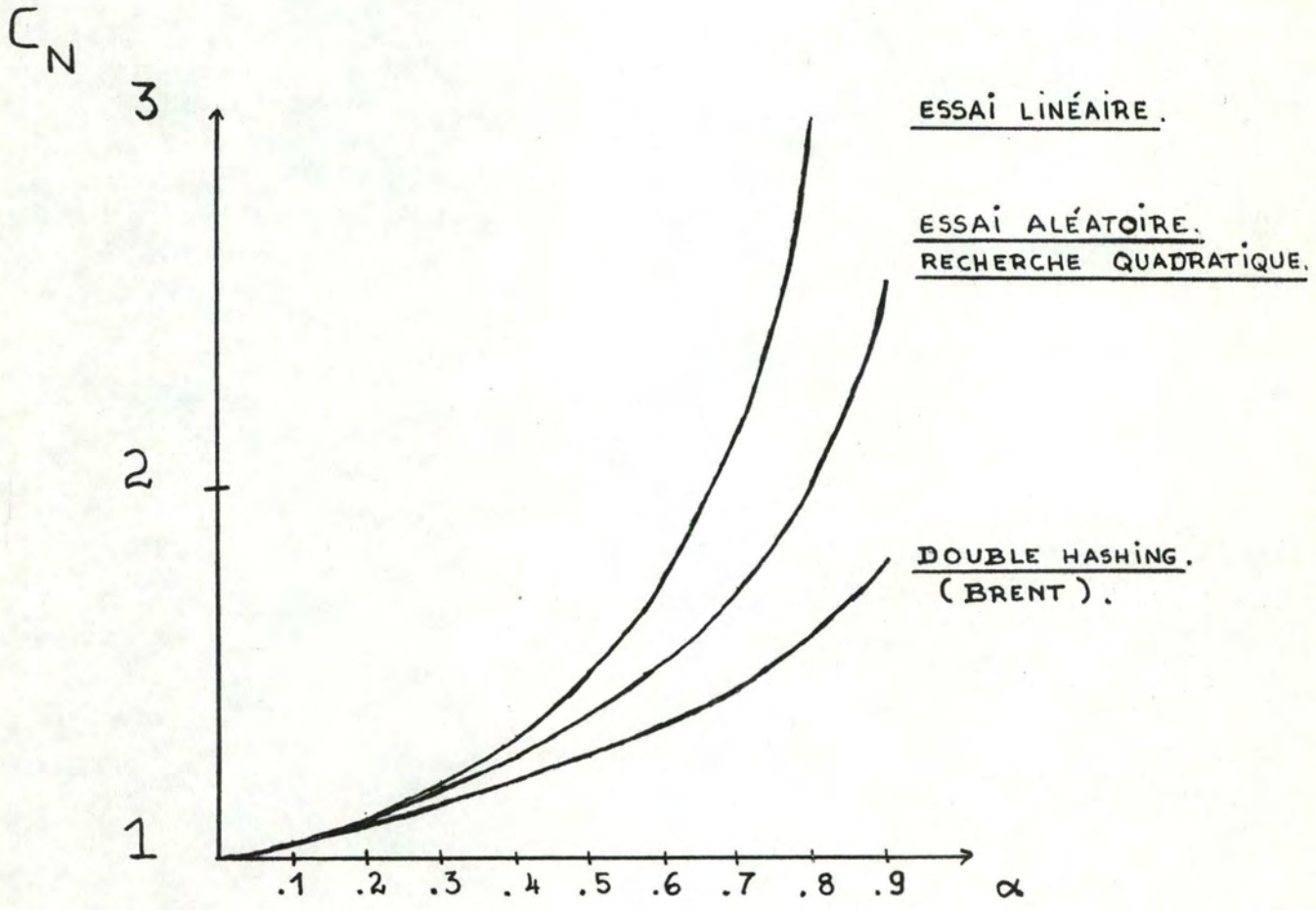


Fig. 3.3.3.2.8.(a) Graphique comparatif - Recherche fructueuse

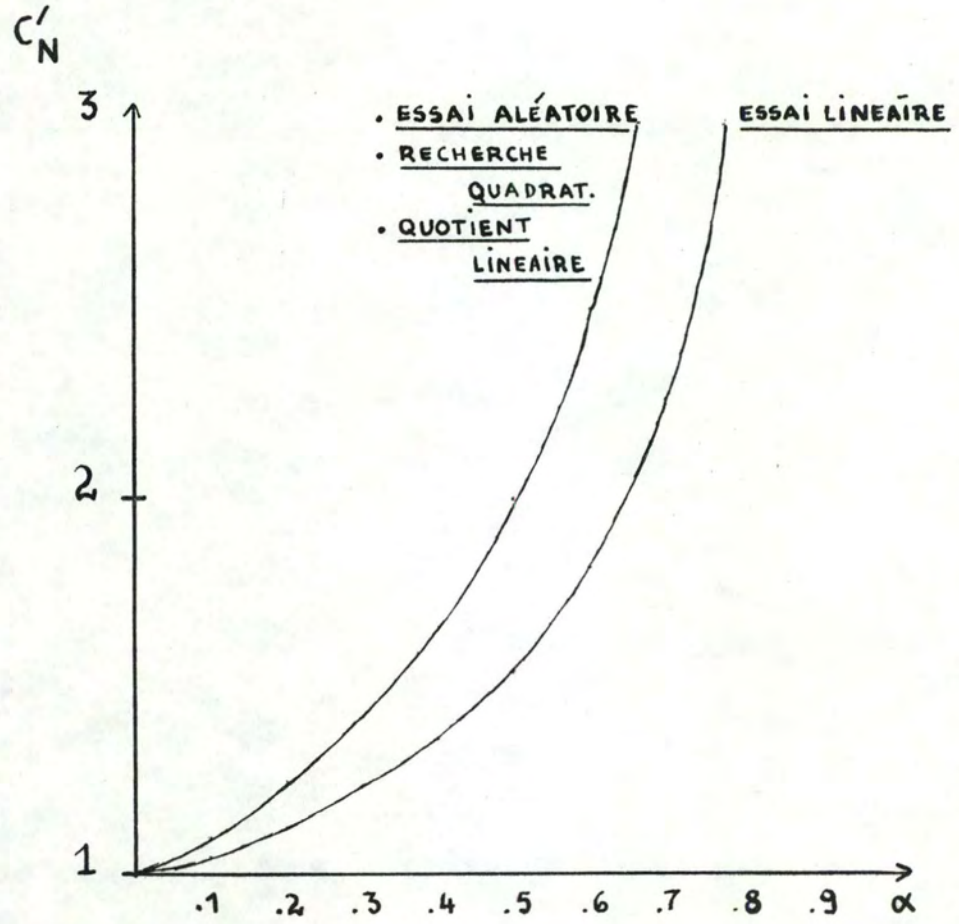


Fig. 3.3.3.2.8.(b) Graphique comparatif - Recherche infructueuse

3.3.3.3. Conclusion sur les méthodes dites de Hash Coding

Concrètement, parmi les méthodes de recherche utilisant le Hash Coding, laquelle préférer ?

Si la contrainte de PLACE-MEMOIRE est absolue, alors, sans l'ombre d'une hésitation, optera-t-on pour une méthode de résolution des collisions sans chaînage; et parmi celles-ci, la méthode du Double Hashing, optimisation de Brent, sera préférée à toutes les autres (1).

Si, au contraire, on recherche un harmonieux compromis PLACE-TEMPS, alors, il apparaît que les méthodes "HASH" avec chaînage l'emportent, et de beaucoup.

Sans doute, la place-mémoire qu'elles requièrent est-elle plus grande et d'autant plus grande que l'enregistrement proprement dit est petit.

Mais non seulement leur temps logique théorique de recherche (fructueuse ou infructueuse) est inférieur, comme le montrent les graphiques récapitulatifs (Fig. 3.3.3.3. a et b), mais encore il est le seul qui ait résisté à la vérification statistique. (2)

Et nous ne prenons même pas en compte le temps de calcul de la fonction F_2.

Chaînage dans la Table ou non ? Si le chaînage se fait hors de la Table - ou encore si F_1 délivre pour valeur le numéro d'une entrée de la Table des pointeurs - la perte de place encourue est plus grande (3).

En revanche, en pareil cas, le temps logique de recherche est légèrement inférieur et l'utilisation de la mémoire est plus souple : les enregistrements peuvent, en effet, être rangés dans des blocs de mémoire non nécessairement successifs.

De tout quoi, il ressort qu'à ce niveau, la réponse dépend du penchant personnel de l'analyste ... Nous serions plutôt partisan du chaînage hors de la Table ...

---

(1) Cfr. p. 141.

(2) Cfr. p. 83.

(3) Cfr. p. 96.

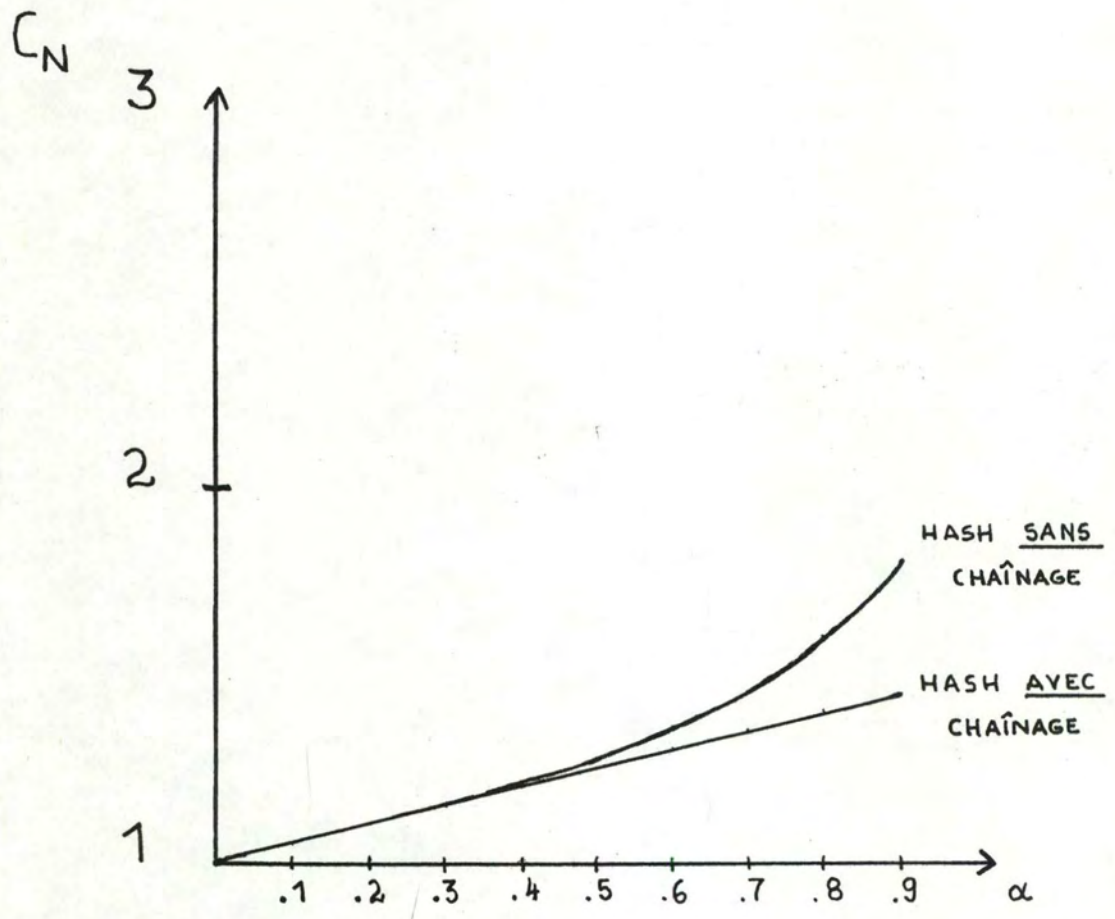


Fig. 3.3.3.3.(a) Graphique récapitulatif - Recherche fructueuse

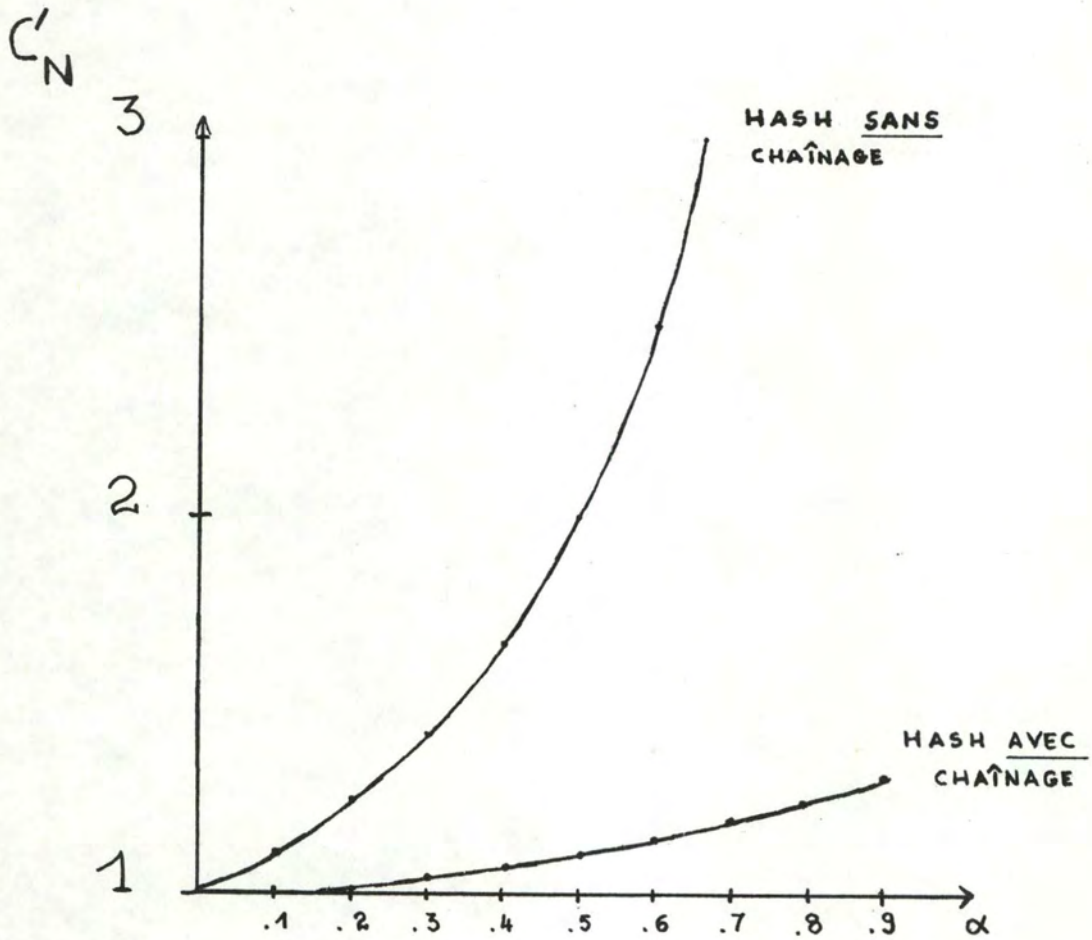


Fig. 3.3.3.3.(b) Graphique récapitulatif - Recherche infructueuse

4. C O N C L U S I O N

Au terme de ce travail, il nous paraît utile de résumer la démarche que nous avons suivie.

Nous nous sommes attaché d'abord à définir le plus complètement possible les termes "Compilateur" et "Table".
Il en est ressorti qu'un programme Compilateur n'était rien d'autre qu'un programme traducteur, une Table un petit fichier muni, comme tout fichier, d'une structure sémantique, d'une structure d'accès et d'une structure d'implémentation physique.
Comme, d'autre part, pour des raisons que nous avons exposées, il nous était loisible de grouper les problèmes relatifs à la structure d'accès et à la structure d'implémentation, le plan général de notre travail s'est articulé autour de deux grands pôles :

1) la nature des informations contenues dans la Table des Symboles;
2) la recherche des informations dans cette Table.

4.1. Nature des Informations

D'abord, nous nous sommes préoccupé de découvrir la méthode selon laquelle devrait être recueillie l'information nécessaire à la résolution du problème de compilation.

Nous avons ensuite appliqué cette méthode au langage ALGOL 60, pour lequel le problème de la compilation se posait dans les termes les plus généraux, à l'exception du cas des variables structurées; nous leur avons, dès lors, réservé un développement particulier.

4.1.1. Application au langage ALGOL 60

Pour chacune des Classes Grammaticales distinguées par le Rapport ALGOL 60, nous avons d'abord recherché les informations minimales que réclamait le processus de compilation.
Ensuite, nous avons discuté de la manière dont il convenait de structurer l'article de la Table des Symboles; il nous est apparu qu'un article à longueur fixe semblait préférable à un article à longueur variable.
Cette conclusion rejoignait, du reste, les résultats des analyses de chercheurs qui, tels Randel, Grau et Gries, avaient conçu un compilateur ALGOL 60 et publié le fruit de leur travail.

4.1.2. Variables structurées

Au niveau de la Table des Symboles, les variables structurées posent deux problèmes : comment implanter la structure dans la Table ? Comment retrouver un élément de la structure dans la Table ?

Nous avons examiné les deux grands types de solutions proposées : solution non-déterministe ou solution déterministe, et nous en avons conclu que, dans le cas normal, la solution non-déterministe paraît de beaucoup la plus simple et la meilleure.

4.2. Recherche des Informations

Après avoir défini ce qu'il fallait entendre, selon nous, par problème de recherche, nous avons rangé en deux grandes classes, les différentes méthodes de recherche par clé : la première regroupe les méthodes qui n'ont recours qu'à une simple comparaison entre les clés; la seconde, celles des méthodes qui utilisent les propriétés digitales des clés (Hash Coding).

4.2.1. Recherche par simple comparaison de clés

Successivement, nous avons décrit le fonctionnement et analysé les performances des méthodes de recherche séquentielle, dichotomique et par arbre binaire.

En conclusion, il nous est apparu que les performances de Temps s'accroissent de façon intolérable au fur et à mesure que le volume du fichier se gonfle.

4.2.2. Recherche par Hash Coding

Pour les méthodes de recherche par Hash Coding, il fallait débattre deux questions : quelle fonction hash retenir ? Quelle méthode de résolution des collisions choisir ?

Nous avons examiné pour chacun de ces deux problèmes les différents types de solutions proposées à ce jour, pour conclure :

1) que la division répond le mieux aux exigences qu'on pouvait formuler à l'endroit d'une fonction hash idéale;
2) que les méthodes de résolution des collisions par chaînage l'emportent, en performances, sur les méthodes de résolution sans chaînage.

Au total, eu égard aux caractéristiques de notre fichier (taux de consultation élevé , articles de petite taille , Mémoire Centrale pour support), il nous a semblé opportun :

- de donner aux articles une longueur fixe;
- de procéder à la consultation du fichier par le truchement des méthodes de résolution par Hash Coding.

Solutions les plus "simples", sans doute, mais non les plus immédiates...

LISTE DES ABREVIATIONS

BELL, Quadratic Quotient = I.R. BELL, The Quadratic Quotient Method : a hash code eliminating secondary clustering, CACM, 13,2 (Fév. 1970), pp. 107 - 109

BELL et KAMAN, Linear Quotient = J.R. BELL et C.H. KAMAN, The linear Quotient hash Code, CACM, 13, 11 (Nov. 1970), pp. 675 - 677.

BRENT, Reducing Time = R.P. BRENT, Reducing the retrieval Time of Scatter Storage Techniques, CACM, 16,2 (Fév. 1973), pp. 105 - 109.

CHERTON, Cours = C. CHERTON, Cours de théorie des fichiers, Namur, 1973.

GATES, Simple Technique = G.W. GATES and D.A. POPLAWSKI, A simple Technique for structured Variable lookup, CACM, 16,9 (Sept. 1973), pp. 561 - 565.

GRAU, Translation = A.A. GRAU, U. HILL, H. LANGMAACK, Translation of ALGOL 60, Springer Verlag, Berlin, 1967.

GRIES, Compiler Construction = D. GRIES, Compiler Construction for digital Computers, Wiley, New-York, 1971.

HOPCROFT, Formal Languages = J.E. HOPCROFT and J.P. ULLMAN, Formal Languages and their Relation to Automata, Addison - Wesley, Reading, Massachusetts, 1969.

HOPGOOD, Compiling Techniques = F.R.A. HOPGOOD, Compiling Techniques, Mac Donald, London, 1969.

INGERMAN, Translator = P.Z. INGERMANN, Syntax oriented Translator, Academic Press, London, 1966.

KNUTH, Art of Programming I = D.E. KNUTH, The Art of Computer Programming, vol. 1, Fundamental Algorithms, Addison - Wesley, Reading, Massachusetts, 2e éd., 1969.

KNUTH, Art of Programming III = D.E. KNUTH, The Art of Computer Programming, vol. 3, Sorting and Searching, Addison Wesley, Reading, Massachusetts, 1973.

LEE, SYMTAB = J.A.N. LEE, The Anatomy of a Compiler, Reinhold book corporation, New-York, 2e éd. 1967.

LEROY, Cours = H. LEROY, Théorie des grammaires formelles, Namur, s.d.

LUM, Transform Techniques = V.Y. LUM, P.S.T. YVEN and M. DODD, KEY-to-Address transform Techniques : a fundamental Performance Study on large existing formatted Files, CACM, 14,4 (Avril 1971), pp. 228 - 238.

MAURER, Improved hash = W.D. MAURER, An improved hash Code for Scatter Storage, CACM, 11, 1 (janv. 1968), pp. 35 - 38.

Mc KEEMAN, Compiler Generator = W.M. Mc KEEMAN, J.J. HORNING and D.B. WORTMAN, A Compiler Generator, Prentice - Hall Inc., Englewood Cliffs N.J., 1970.

MINSKY, Computation = M.L. MINSKY, Computation, London.

MORRIS, Scatter Storage = R. MORRIS, Scatter Storage Techniques, CACM, 11,1 (Janv. 1968), pp. 38 - 44.

PETERSON, Random-access = W.W. PETERSON, Addressing for random-acces Storage, IBM Journal Research and Development 1, (1957), pp. 130 - 146.

RADKE, Quadratic residue = C.E. RADKE, The Use of quadratic Residue Research, CACM, 13,2 (Fév. 1970), pp. 103 - 105.

RANDEL, Implementation = B. RANDEL and L.R.J. RUSSEL, ALGOL 60 Implementation, the Translation and Use of ALGOL 60 Programs on a Computer, Academic Press, London and New-York, 1964.

Rapport ALGOL 60 = Revised Report on the Algorithmic Language ALGOL 60, CACM, 6, 1 (Janv. 63) pp. 1 -17.

WEGNER, Programming Languages = P. WEGNER, Programming Languages, information structures and machine organization, Mc Graw-Hill, New-York, 1968.